

第 6 章 GDI+

GDI+(Graphics Device Interface Plus 图形设备接口加)是 Windows XP 和 Windows Server 2003 操作系统的子系统，也是.NET 框架的重要组成部分，负责在屏幕和打印机上绘制图形图像和显示信息。

顾名思义，GDI+是 Windows 早期版本所提供的图形设备接口 GDI 的后续版本。GDI+是一种应用程序编程接口(API)，通过一套部署为托管代码的类来展现。这套类被称为 GDI+的“托管类接口”。

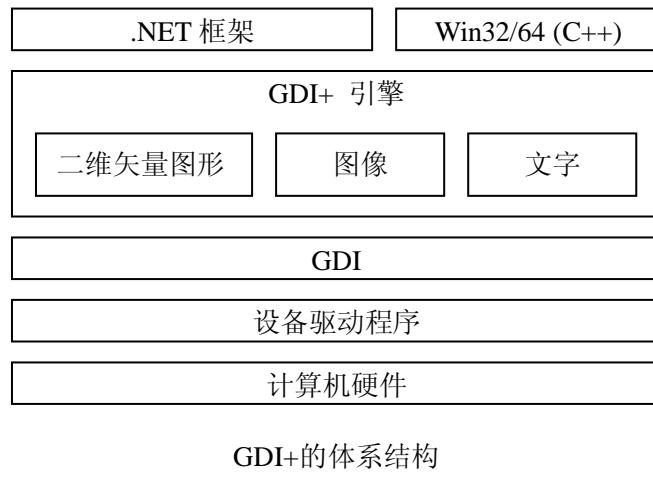
GDI+最早是于 2001 年随 Windows XP 一起推出的一种 API，后来又被包装进.NET 框架的托管类库之中，成为.NET 中窗体绘图的主要工具。

GDI+不但在功能上比 GDI 要强大很多，而且在代码编写方面也更简单，因此会很快成为 Windows 图形图像程序开发的首选。

本章将介绍 GDI+的特点和新增功能，以及 GDI+ API 的具体使用方法，包括二维矢量图形的绘制、图像处理的应用、以及文字的显示。

6.1 概述

GDI+与 GDI 一样，都具有设备无关性。应用程序的程序员可利用 GDI+这样的图形设备接口在屏幕或打印机上显示信息，而不需要考虑特定显示设备的具体情况。应用程序的程序员调用 GDI+类提供的方法，而这些方法又反过来相应地调用特定的设备驱动程序。GDI+将应用程序与图形硬件隔离，而正是这种隔离允许开发人员创建设备无关的应用程序。



GDI+的体系结构

本节首先介绍 GDI+的几个主要新增的特性及其功能，然后说明它给 Windows 图形图像程序的开发模式带来的变化，最后给出一个代码实例，介绍如何在 VC++中使用 GDI+进行程序开发。

1. GDI+的功能

GDI+主要提供了以下三种功能：

1) 二维矢量图形

矢量图形包括坐标系统中的系列点指定的绘图基元（如直线、曲线和图形）。例如，直线可通过它的两个端点来指定，而矩形可通过确定其左上角位置的点并给出其宽度和高度的一对数字来指定。简单路径可由通过直线连接的点的数组来指定。贝塞尔样条是由四个控制点指定的复杂曲线。

GDI+提供了存储基元自身相关信息的类（结构）、存储基元绘制方式相关信息的类，以及实际进行绘制的类。例如，`Rectangle` 结构存储矩形的位置和尺寸；`Pen` 类存储有关线条颜色、线条粗细和线型的信息；而 `Graphics` 类具有用于绘制直线、矩形、路径和其它图形的方法（类似于 GDI 中的 `CDC` 类）。还有几种 `Brush` 类，它们存储有关如何使用颜色或图案来填充封闭图形和路径的信息。

用户可以在图元文件中记录矢量图像（图形命令的序列）。GDI+提供了 `Metafile` 类，可用于记录、显示和保存图元文件。`MetafileHeader` 和 `MetaHeader` 类允许您检查图元文件头中存储的数据。

2) 图像处理

某些种类的图片很难或者根本无法用矢量图形技术来显示。例如，工具栏按钮上的图片和显示为图标的图片就难以指定为直线和曲线的集合。拥挤的棒球运动场的高分辨率数字照片会更难以使用矢量技术来制作。这种类型的图像可存储为位图，即代表屏幕上单个点颜色的数字数组。

GDI+提供了 `Image`、`Bitmap` 和 `Metafile` 类，可用于显示、操作和保存位图。它们支持众多的图像文件格式，还可以进行多种图像处理的操作。

3) 文字显示版式

就是使用各种字体、字号和样式来显示文本。GDI+为这种复杂任务提供了大量的支持。GDI+中的新功能之一是子像素消除锯齿，它可以使文本在 LCD 屏幕上呈现时显得比较平滑。

4) 功能汇总

GDI+的 C++ 封装包含 54 个类、12 个函数、6 类（226 个）图像常量、55 种枚举和 19 种结构。GDI+的托管类接口则包含大约 60 个类、50 个枚举和 8 个结构。这两种封装中的 `Graphics` 类都是 GDI+的核心功能，它是实际绘制直线、曲线、图形、图像和文本的类。通过这些类和接口可以实现：

- 使用笔绘制线条和形状
- 使用刷填充形状
- 使用图像、位图和图元文件
- α 混合线条和填充
- 字体和文本
- 构造并绘制曲线
- 用颜色渐变的梯度刷填充形状
- 构造并绘制轨迹
- 变换
- 图形容器
- 区域
- 重新着色
- 读取元数据

等非常丰富强大的功能。

2. GDI+新增特性

1) 渐变画刷

渐变画刷（gradient brush 梯度刷）通过提供用于填充图形、路径和区域的线性渐变画笔和路径渐变画笔，GDI+扩展了 GDI 的功能。渐变画笔还可用于绘制直线、曲线和路径。线性渐变画笔可用于使用颜色来填充图形，画笔在图形中移动时，颜色会逐渐改变。例如，假定通过指定图形左边为蓝色、右边为绿色，创建了一个水平渐变画笔。当用水平渐变画笔填充该图形时，随着画笔从图形的左边移至右边，颜色就会由蓝色逐渐变为绿色。用类似方法定义的垂直渐变画笔填充的图形，颜色从上到下变化。图 6-1 显示了用水平渐变画笔填充的椭圆和用斜式渐变画笔填充的区域。

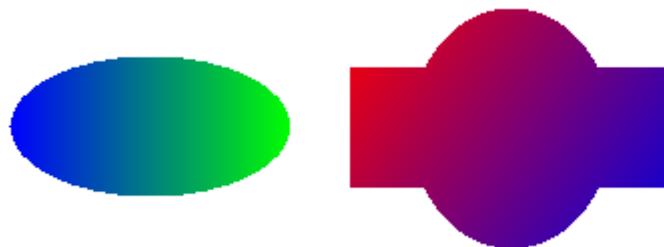


图 6-1 水平和斜式渐变画笔

用路径渐变画笔填充图形时，可选择不同的方法来指定当从图形的一部分至另一部分移动画笔时颜色的变化方式。一种选择是指定中心颜色和边缘颜色，在从图形中间向外边缘移动画笔时，像素逐渐从一种颜色变化到另一种颜色。图 6-2 显示了用路径渐变画笔填充的路径（该路径是用一对贝塞尔样条创建的）。

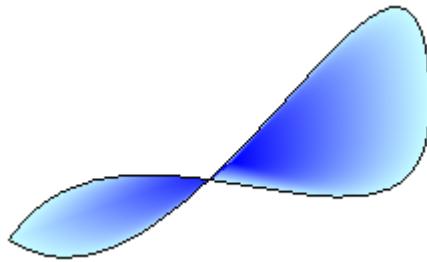


图 6-2 路径渐变画笔

2) 基数样条函数

GDI+支持在 GDI 中不支持的基数样条 (cardinal splines)。基数样条是一连串单独的曲线，这些曲线连接起来形成一条较长的光滑曲线。样条由点的数组指定，并通过该数组中的每一个点。基数样条平滑地（没有锐角）通过数组中的每一个点，因此，比通过连接直线创建的路径更光滑精准。图 6-3 显示了两个路径：一个以基数样条的形式创建；另一个通过连接直线创建。

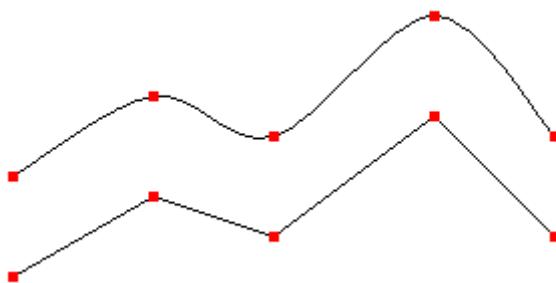


图 6-3 基数样条路径和折线路径

3) 持久路径对象

在 GDI 中，路径属于设备上下文，并且会在绘制时被毁坏。利用 GDI +，绘图由 Graphics 对象执行，可以创建并维护几个与 Graphics 对象分开的持久的路径对象 (persistent path object) —— GraphicsPath 对象。绘图操作不会破坏 GraphicsPath 对象，因此可以多次使用同一个 GraphicsPath 对象来绘制路径。

4) 变换和矩阵对象

GDI+提供了 Matrix (矩阵) 对象，它是一种可以使（缩放、旋转和平移等）变换 (transformation) 简易灵活的强大工具。矩阵对象一般与变换对象联合使用。例如，GraphicsPath 对象具有 Transform 方法，此方法接收 Matrix 对象作为参数。单一的 3×3 矩阵可存储一种变换或一个变换序列。图 6-4 显示了一个路径在执行两种变换前后的情况。

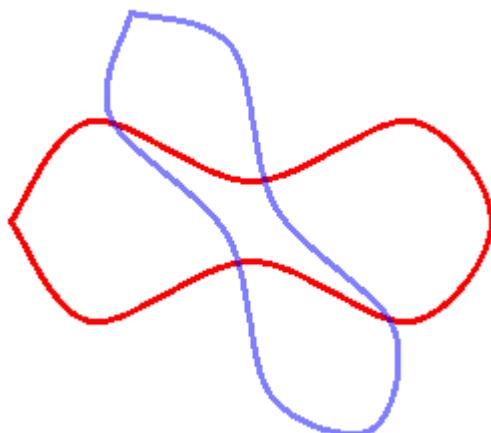


图 6-4 路径的变换

5) 可伸缩区域

GDI+ 通过对可伸缩区域（Scalable Regions）的支持极大地扩展了 GDI。在 GDI 中，区域被存储在设备坐标中，而且，可应用于区域的惟一变换是平移。而 GDI+ 在全局坐标中存储区域，并且允许区域发生任何可存储在变换矩阵中的变换（如缩放和旋转）。图 6-5 显示一个区域在执行三种变换（缩放、旋转和平移）前后的情况。

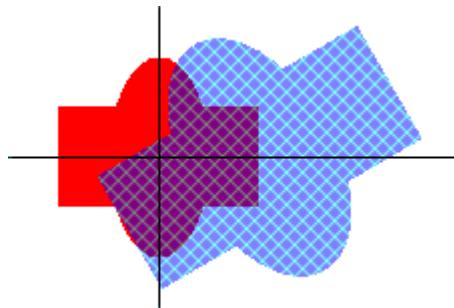


图 6-5 区域的三种变换（缩放、旋转和平移）

6) α 混色

在图 6-5 中，可以在变换区域（用蓝色阴影画笔填充）中看到未变换区域（用红色填充）。这是由 GDI+ 支持的 α 混色（Alpha Blending，透明混合）实现的。使用 α 混色，可以指定填充颜色的透明度。透明色与背景色相混合——填充色越透明，透出的背景色就越多。图 6-6 显示四个用相同颜色（红色）填充、但透明层次不同的椭圆。

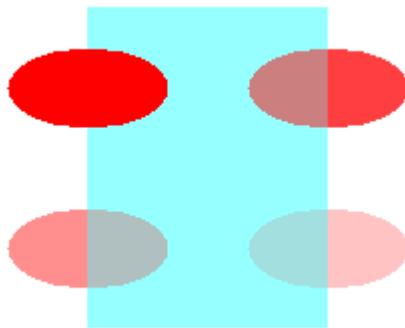


图 6-6 不同透明度

7) 丰富的图像格式支持

GDI+提供 Image、Bitmap 和 Metafile 类，可以用不同的格式加载、保存和操作图像。GDI+支持 BMP、GIF、JPEG、EXIF、PNG、TIFF、ICON、WMF、EMF 共 9 种常见的图像格式。

8) GDI+的不足

虽然，相对于 GDI 来说，GDI+ 确实增加了许多新特性，而且功能更强大，使用也更方便。但是，这并不等于 GDI+ 就能够完全代替 GDI。

因为 GDI+实际上 GDI+ 是 GDI 的封装和扩展，GDI+的执行效率一般要低于 GDI 的。另外，GDI+不支持图的位运算，那么就不能进行异或绘图等操作。而且在 VC 中，GDI+ 还不支持双缓存机制（如内存 DC 和显示 DC），这将大大影响 GDI+ 在高速图形、图像、动画和视频等方面的应用。

3. 编程模式的改变

GDI+的出现，也使基于 GDI 的编程模式产生了很大变化：GDI+用一个“无状态模式”，取代了 GDI 中（需要先将各种工具和项目选入 DC 对象后，才能进行绘图的）“状态模式”。

主要体现在以下几个方面：

1) DC 句柄和图形对象

设备上下文（DC = Device Context）是 GDI 中使用的一种结构，用于存储与特定显示设备的功能、以及指定如何在该设备上绘制项目之属性相关的信息。用于屏幕显示的 DC 还与特定窗口相关联。为了使用 GDI API 进行绘图，必须首先获得一个 DC 的句柄（HDC = Handle to a DC），然后将该句柄作为参数，传递给实际进行绘图的 GDI 函数。还可以将此句柄作为参数，传递给获取和设置 DC 属性的 GDI 函数。

使用 GDI+，不需要再（直接）使用句柄或设备上下文，而是只需（通过 HDC）创建一个 Graphics 对象，然后用熟悉的面向对象方式来调用其中的各种绘图方法，例如：

```
myGraphicsObject.DrawLine(&pen, x1, y1, x2, y2);
```

正如 DC 是 GDI 的核心，Graphics 对象也位于 GDI+的核心。DC 和 Graphics 对象的作用相似，但在使用设备上下文 (GDI) 的基于句柄的编程模式和使用 Graphics 对象 (GDI+) 的面向对象的编程模型之间，存在一些基本的差异。

Graphics 对象（像 DC 一样）与屏幕上的特定窗口关联，并具有指定如何绘制项目的属性（如 SmoothingMode 和 TextRenderingHint）。但是，Graphics 对象不受笔、刷、路径、图像或字体的约束，这与设备上下文不同。例如，使用设备上下文绘制线条之前，必须先调用 SelectObject 以使笔对象和 DC 关联，即将笔选入 DC 中。在设备上下文中绘制的所有线条均使用该笔，直到选择另一支不同的笔为止。在 GDI+中，将 Pen 对象作为参数传递给 Graphics 类的 DrawLine 方法。可以在一系列的 DrawLine 调用的每个调用中，使用不同的 Pen 对象，而不必将给定的 Pen 对象与 Graphics 对象关联。

2) 绘制线条的两种方法

下面每个示例都从点(20, 10)到点(200, 100)绘制一条宽为 3 的红色线条。第一个示例调用 GDI，第二个示例则通过托管类接口调用 GDI+；它们都有分别使用 API 和 MFC 的两个版本。

(1) 用 GDI 画线

● API

要使用 GDI 绘制线条，需要两个对象：设备上下文和笔。在 WM_PAINT 的消息响应代码中，通过调用 BeginPaint，可以获得设备上下文句柄；通过调用 CreatePen，则可以获得笔句柄。再调用 SelectObject 以将笔选入设备上下文。调用 MoveToEx，将笔的当前位置设在(20, 10)，然后调用 LineTo，在笔的当前位置与位置(200, 100) 之间绘制一条线条。请注意，所有这些函数和类型，都是全局的。而且 MoveToEx 和 LineTo 均将 hdc (设备上下文的句柄) 作为参数接收。

WM_PAINT:

```
HDC hdc; // DC 句柄  
PAINTSTRUCT ps; // 点结构  
HPEN hPen; // 笔句柄  
HPEN hPenOld; // 用于保存原笔的句柄  
hdc = BeginPaint (hWnd , &ps); // 获得 DC 句柄，开始绘制，其中 hWnd 为窗口句柄  
hPen = CreatePen(PS_SOLID, 3, RGB(255, 0, 0)); // 创建红色画笔，宽 3  
hPenOld = SelectObject(hdc, hPen); // 选笔入 DC  
MoveToEx(hdc, 20, 10, NULL); // 最后一个参数是返回用的旧当前点的结构指针  
LineTo(hdc, 200, 100); // 画线  
SelectObject(hdc, hPenOld); // 选原笔入 DC  
DeleteObject(hPen); // 删除创建的笔  
EndPaint(hWnd, &ps); // 绘制结束  
break;
```

● MFC

利用 MFC 进行 GDI 绘图，步骤与 API 的差不多，只是 MFC 将各种 GDI 功能封装到了不同的类中。例如，笔的类为 CPen、点的类为 CPoint、设备上下文的类为 CDC。而且所有的绘图函数都被封在 CDC 类中，所以只能作为其对象的成员函数才能被使用，当然也就不需要再带 HDC 句柄作为输入参数了。

```
void CGdipDemoView::OnDraw(CDC* pDC) {  
    CGdipDrawDoc* pDoc = GetDocument();  
    ASSERT_VALID(pDoc);  
    if (!pDoc) return;  
    // TODO: 在此处为本机数据添加绘制代码  
    CPen pen(PS_SOLID, 3, RGB(255, 0, 0)); // 创建红色画笔，宽 3  
    pDC->SelectObject(&pen); // 选入 DC  
    // pDC->SelectObject(new CPen(PS_SOLID, 3, RGB(255, 0, 0))); // 上两步可以  
    合并  
    pDC->MoveTo(20, 10); // 将当前点移到直线的起点  
    pDC->LineTo(200, 100); // 画线  
}
```

(2) 用 GDI+画线

● API

使用 GDI+和托管类接口绘制线条，需要 Graphics 对象和 Pen 对象。绘制线条涉及调用 Graphics 类的 DrawLine 方法。DrawLine 方法的第一个参数是 Pen 对象。与前面 GDI 示例中显示的技术（将笔选入设备上下文）相比，这是一个更加简单而灵活的方案。

WM_PAINT:

```
HDC hdc;  
PAINTSTRUCT ps;  
Pen *myPen;  
Graphics *myGraphics;  
hdc = BeginPaint(hWnd, &ps);  
myPen = new Pen(0xffff0000, 3); // 创建一个笔，宽 3，红色  
myGraphics = new Graphics(hdc); // 利用 DC 句柄创建图形对象  
myGraphics->DrawLine(myPen, 20, 10, 200, 100); // 调用图形对象的画线方法  
EndPaint(hWnd, &ps);  
break;
```

● MFC

利用 MFC 进行 GDI+绘图，步骤与 API 的差不多。只是代码改在 OnDraw 函数中，而且获取 DC 句柄的方法不同。

```
void CGdipDemoView::OnDraw(CDC* pDC) {  
    CGdipDrawDoc* pDoc = GetDocument();  
    ASSERT_VALID(pDoc);  
    if (!pDoc) return;  
    // TODO: 在此处为本机数据添加绘制代码  
    Graphics myGraphics(pDC->m_hDC); // 利用 DC 句柄创建图形对象  
    Pen myPen(Color(255, 0, 0), 3); // 创建一个笔，宽 3，红色
```

```

myGraphics.DrawLine(&myPen, 20, 10, 200, 100); // 调用图形对象的画线方法
// 上两步也可以合并:
// myGraphics.DrawLine(&Pen(Color(255, 0 , 0), 3), 20, 10, 200, 100);
}

```

3) 作为参数的笔、刷、路径、图像和字体

前面的示例显示：创建和维护 Pen 对象可以与提供绘制方法的 Graphics 对象分开。创建和维护 Brush、GraphicsPath、Image 和 Font 对象也可以与 Graphics 对象分开，Graphics 类提供的许多绘制方法都将这些对象作为参数接收。

例如，Brush 对象作为参数传递至 FillRectangle 方法，GraphicPath 对象作为参数传递至 DrawPath 方法。同样，Image 和 Font 对象传递至 DrawImage 和 DrawString 方法。这与 GDI 不同，在 GDI 中，需要将笔、刷、路径、图像或字体选入 DC，然后将 DC 的句柄作为参数传递至绘制函数或采用 CDC 类对象的函数来绘图。

4) 方法重载

许多 GDI+方法都是重载的，即，若干方法共享同一名称，却有不同的参数列表。这一点与用 MFC 封装后的 GDI 类似，但是 GDI+中的重载方法要更多一些。（注意，在 .NET、C#、Java 和 VB 中，都把类的成员函数称为方法。当我们在 C++中，使用.NET 框架类库中的类和功能时，也常常将其成员函数改称为方法。）例如，画线的重载方法有：

```

Status DrawLine(const Pen* pen, REAL x1, REAL y1, REAL x2, REAL y2);
Status DrawLine(const Pen* pen, const PointF& pt1, const PointF& pt2);
Status DrawLine(const Pen* pen, INT x1, INT y1, INT x2, INT y2);
Status DrawLine(const Pen* pen, const Point& pt1, const Point& pt2);

```

其中，
`typedef int INT;` `class Point {public: INT X; INT Y;};`
`typedef float REAL;` `class PointF {public: REAL X; REAL Y;};`

5) 无当前位置

前面所述的 DrawLine 方法中显示：线条的起点和终点均被作为参数接收。这与 GDI 方案不同，在 GDI 中，调用 MoveToEx(hdc, x1, y1, NULL)或 pDC->MoveTo(x1, y1)来设置当前笔位置之后，再调用 LineTo (hwnd , x2 , y2)或 pDC->LineTo(x2, y2)来绘制一条从(x1, y1) 到 (x2 , y2) 的线条。GDI+从总体上已经放弃了当前位置的概念。

6) 绘制和填充的不同方法

论及绘制轮廓和填充图形内部时，GDI+要比 GDI 更灵活。GDI 有一个 Rectangle 函数，可一步完成绘制轮廓和填充矩形内部。轮廓由当前选定的笔绘制，而内部则由当前选定的刷来填充。

GDI+使用不同的方法来绘制轮廓和填充矩形内部。Graphics 类的 DrawRectangle 方法将

Pen 对象作为其参数之一，而 FillRectangle 方法将 Brush 对象作为其参数之一。

7) 构造区域

GDI 提供几种用于创建区域的函数(在 MFC 中，它们被封装在 CRng 类里): CreateRectRgn、CreateEllipticRgn、CreateRoundRectRgn、CreatePolygonRgn 和 CreatePolyPolygonRgn。您或许希望 GDI+中的 Region 类也有类似的构造函数，将矩形、椭圆、圆角矩形和多边形作为参数接收，但事实并非如此。GDI+中的 Region 类提供一个接收 Rectangle 对象的构造函数和另一个接收 GraphicsPath 对象的构造函数。如果想基于椭圆、圆角矩形或多边形构造区域，可以通过创建一个 GraphicsPath 对象（可包含椭圆的对象等），然后将其传递至 Region 构造函数来轻松实现。

GDI+通过组合图形和路径，使得构成复杂区域十分简单。Region 类具有 Union 和 Intersect 方法，可用于扩展具有路径的现有区域或其它区域。GDI+方案一个很好的功能就是 GraphicsPath 对象在作为参数传递至 Region 构造函数时不会被破坏（在 GDI 中，可以使用 PathToRegion 函数将路径转换为区域，但在此过程中，路径将被破坏）。另外，GraphicsPath 对象在作为参数传递给 Union 或 Intersect 方法时也不会被破坏，因此，在一些单独的区域中，可以将给定的路径作为构造块使用。例如：

```
Region region1(rect1);
Region region2(rect2);
region1.Union(onePath);
region2.Intersect(onePath);
```

4. GDI+的使用

下面通过一个简单的例子，来说明如何使用 GDI+进行应用程序开发。

1) GDI+开发包

若采用的是 Visual C++ 2005，则已经包含了开发 GDI+应用程序所需的所有东西。包括：

- 动态链接库文件 gdiplus.dll
- 静态链接库文件 gdiplus.lib
- 代码中所需要的头文件 gdiplus*.h
- 帮助文档 gdicpp.chm 和 gdicpp.chi

如果你使用的操作系统是 Windows XP 或 Windows Server 2003，则 GDI+所对应的动态链接库，已经被包含在其中。gdiplus.dll 一般位于操作系统的 WinSxS (Windows side-by-side assembly, 视窗并行程序集) 目录中，例如：

C:\WINDOWS\WinSxS\x86_Microsoft.Windows.GdiPlus_6595b64144ccf1df_1.0.0.0_x-ww_8d

353f13\gdiplus.dll (1661KB, 2002.10.8)

C:\WINDOWS\WinSxS\x86_Microsoft.Windows.GdiPlus_6595b64144ccf1df_1.0.2600.2180_x-

ww_522f9f82\gdiplus.dll (1672KB, 2004.8.4)

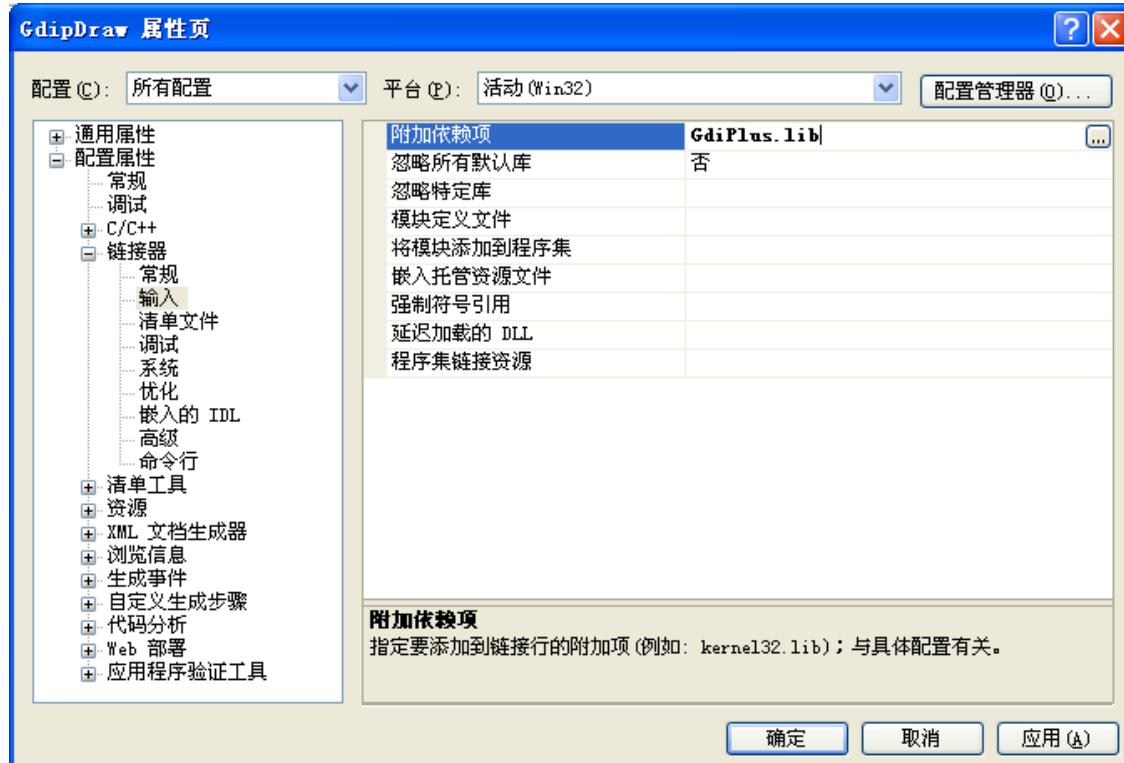
而 GDI 的动态链接库 gdi32.dll，却一般在操作系统的 32 位系统目录中：

F:\WINDOWS\system32\gdi32.dll (272KB, 2004.8.4)

如果开发工具采用 Visual C++ 6.0，而且操作系统是 Windows 2000，则需要安装 GDI+的开发包。如果你已经安装了.NET 框架，则其中已经包含了该开发包。如果还没有安装，则需要自己去微软的网站免费下载 GDI+开发包或.NET 框架(可能需要先通过微软的正版操作系统软件验证)。

2) VC 中的设置

- 在 VS05 中，选“项目/*属性”菜单项，打开项目的属性页窗口，先选“所有配置”，再选“配置属性/链接器/输入”项，在右边上部的“附加依赖项”栏的右边，键入 GdiPlus.lib (参见下图) 后按“应用”钮，最后按“确定”钮关闭对话框。



- 在要使用 GDI+的文件(如视图类的头文件或代码文件)头部包含 GDI+的头文件：

```
#include <gdiplus.h>
```

并加上使用 GDI+命名空间的 using 指令(区分大小写，注意首字母大写)：

```
using namespace Gdiplus;
```

下面是 VC05 中，GDI+头文件和动态链接库文件，缺省所在的目录：

C:\Program Files\Microsoft Visual Studio 8\VC\PlatformSDK\include\GdiPlus*.h

C:\Program Files\Microsoft Visual Studio 8\VC\PlatformSDK\Lib\GdiPlus.lib

下面是 VC05 中，GDI 头文件和动态链接库文件，缺省所在的目录：

C:\Program Files\Microsoft Visual Studio 8\VC\PlatformSDK\include\WinGDI.h (API)

C:\Program Files\Microsoft Visual Studio 8\VC\atlmfc\include\afxwin.h (MFC)

C:\Program Files\Microsoft Visual Studio 8\VC\PlatformSDK\Lib\Gdi32.lib

3) 存在的问题

另外，VC05 与 GDI+存在一些问题，例如：

(1) 重画问题

GDI+程序往往在窗口被创建时，不能自动重画（没有自动调用 OnDraw 函数）。解决办法是，在创建图形对象后，自己调用视图类（基类 CWnd）的成员函数 RedrawWindow：

```
BOOL RedrawWindow(LPCRECT lpRectUpdate = NULL, CRgn* prgnUpdate = NULL,
    UINT flags = RDW_INVALIDATE | RDW_UPDATENOW | RDW_ERASE);
```

其中，lpRectUpdate 为窗口客户区中需要重画的矩形（NULL 表示整个客户区矩形重画）、prgnUpdate 表示需要重画的区域（NULL 表示整个客户区矩形区域重画）、flags 为特征标志（RDW_INVALIDATE 指定范围无效、RDW_UPDATENOW 立即更新、RDW_ERASE 擦除背景）。

例如：

```
Graphics graph(pDC->m_hDC);
RedrawWindow(); // 一般输入参数缺省值即可
// 相当于 Invalidate(); UpdateWindow(); 的综合效果
```

注意：不能在 OnDraw 和 OnPaint 函数中调用 RedrawWindow，那样会造成反复调用，产生死循环。

其实，只要 GDI+的两个初始化语句放置的位置对（必须放在 CWinApp::InitInstance ()；语句之前，参见下面 4) 中的说明），就不会出现该问题。

(2) new 问题

不能使用 new 来动态创建 GDI+对象。解决办法是（我摸索出的，不一定最好），打开（缺省）位于 C:\Program Files\Microsoft Visual Studio 8\VC\PlatformSDK\Include 目录中的 GdiplusBase.h 头文件，并注释掉里面 GdiplusBase 类的内容（该类其实只含 new、new[]、delete 和 delete[] 这四个运算符的重载），使其成为一个空类（不要删除整个类）。

对实验室中的写保护机器，不能修改安装目录中的 GdiplusBase.h 头文件，解决办法是：

- 将该头文件复制到你的项目目录中；
- 注释掉该头文件里面 GdiplusBase 类的内容（保留类定义）；
- 在你项目中所有的#include <gdiplus.h> 语句之前，包含"GdiplusBase.h"头文件，形如：

```
#include "gdiplusBase.h"
#include <gdiplus.h>
```

- 则编译系统会优先包含项目目录中的 gdiplusBase.h 头文件，从而屏蔽掉原来位于平台 SDK 的 Include 目录中的同名头文件。

你也可以在有些使用 new 的地方改用&，例如你可以将代码

```
Pen *pPen = new Pen(Color::Red);
```

改为

```
Pen *pPen = &Pen(Color::Red);
```

又例如，你也可以将代码：

```
graphics.DrawPolygon(new Pen(Color::Green), points, n);
```

改为

```
Pen pen(Color::Green);
graphics.DrawPolygon(&pen, points, n);
```

或直接改为

```
graphics.DrawPolygon(&Pen(Color::Green), points, n);
```

(3) 调试问题

因为现在版本的 VC05 存在许多 Bug，特别是 GDI+程序在调试时的问题就更多。解决办法是：

- 在编译运行时，不使用 Debug 配置，而改用 Release 配置；
- 运行时不使用调试运行（F5），而改用不调试直接运行（Ctrl +F5）；
- 最好是用静态链接的 MFC 库，而不用 DLL 动态库。

常用的调试方法有：

- 使用 MessageBox 信息框：
 - 在视图类中的常用格式为

```
MessageBox(L"提示信息");
```
 - 在应用程序类和文档类中的常用格式为

```
MessageBox(NULL, L"提示信息", L"标题", MB_OK);
```
- 设置断点，然后逐步运行（F10）
- 运行当前位置，然后逐步运行（F10）
- 利用调试界面中的“局部变量”和“监视 1”等窗口，来查看变量当前的值

4) 用 MFC 开发 GDI+程序

创建一个名为 GDIPlusDemo 的 MFC 单文档应用程序项目。

首先要进行 GDI+系统的初始化，这需要在应用程序类 CGDIPlusDemoApp 中声明一个成员变量：

```
ULONG_PTR m_gdiplusToken; // ULONG PTR 为 int64 类型
```

并在该类的初始化函数 CGDIPlusDemoApp::InitInstance() 中加入以下代码来对 GDI+进行初始化：

```
GdiplusStartupInput gdiplusStartupInput;
GdiplusStartup(&m_gdiplusToken, &gdiplusStartupInput, NULL);
```

注意：这两个语句必须加在应用程序类的 InitInstance 函数中的

```
CWinApp:: InitInstance();
```

语句之前，不然以后会造成视图窗口不能自动重画、程序中不能使用字体等等一系列问题。

还要在 CGDIPlusDemoApp::ExitInstance() 函数中加入以下代码来关闭 GDI+：

```
GdiplusShutdown(m_gdiplusToken);
```

上面的 InitInstance 和 ExitInstance 都是应用程序类的重写型成员函数。而且，缺省时无 ExitInstance，需要自己利用属性窗口来添加（不要手工添加）。

接下来就可以利用 GDI+进行绘图了。

在 OnDraw 函数中画图：

```
CGDIPlusDemoView::OnDraw (CDC* pDC) {  
    Graphics graph(pDC->m_hDC); // 创建图形对象  
    Pen bluePen(Color(255, 0, 0, 255)); // 创建蓝色笔  
    Pen redPen(Color(255, 255, 0, 0)); // 创建红色笔  
    int y = 255; // y 的初值  
    for (int x = 0; x < 256; x += 5) { // 绘制红蓝网线  
        graph.DrawLine(&bluePen, 0, y, x, 0);  
        graph.DrawLine(&redPen, 255, x, y, 255);  
        y -= 5;  
    }  
    for (y = 0; y < 256; y++) { // 画绿色透明度水平渐变的线 (填满正方形)  
        Pen pen(Color(y, 0, 255, 0)); // A green pen with shifting alpha  
        graph.DrawLine(&pen, 0, y, 255, y);  
    }  
    for (int x = 0; x < 256; x++) { // 画品红色透明度垂直渐变的线 (填满扁矩形)  
        Pen pen(Color (x, 255, 0, 255)); // A magenta pen with shifting alpha  
        graph.DrawLine(&pen, x, 100, x, 200);  
    }  
}
```

运行的结果如图 6-7 所示。(其中，第一个图为第一个循环所绘制的结果、第二个图为前两个循环所绘制的结果、第三个图为全部循环所绘制的结果)

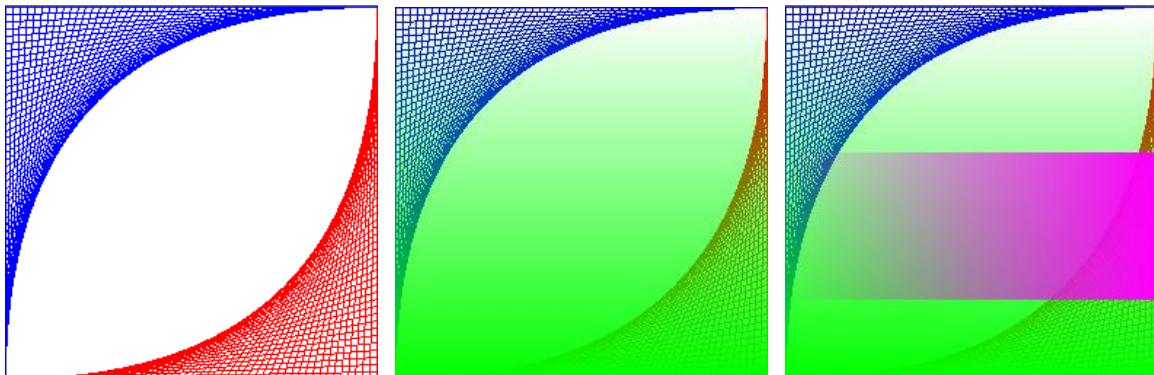


图 6-7 透明度的连续变化

5) GDI+帮助文档

GDI+的英文帮助内容,位于 VS05 的“目录/Win32 和 COM 开发/Graphics and Multimedia/GDI+”，主要的参考资料位于其子目录“GDI+ Reference”中。

GDI+的中文帮助内容位于 VS05 的“目录/开发工具和语言/Visual Studio 文档/基于 Windows 的应用程序、组件和服务/创建基于 Windows 的应用程序/Windows 窗体 (Windows Forms) /增强 Windows 窗体应用程序/Windows 窗体中的图形和绘制”，其中包括“图形概述 (Windows 窗体)”、“关于 GDI+ 托管代码”和“使用托管图形类”三个子目录。

也可以键入下列地址直接进入：

- 英文帮助：

<ms-help://MS.VSCC.v80/MS.MSDN.v80/MS.WIN32COM.v10.en/gdicpp/GDIPlus/GDIPlus.htm>

- 中文帮助:

ms-help://MS.VSCC.v80/MS.MSDN.v80/MS.VisualStudio.v80.chs/dv_fxmcigrnl/html/362532c5-1a06-4257-bdc8-723461009ede.htm

5. GDI+的组成

GDI+ API 包含 54 个类、12 个函数、6 类 (226 个) 图像常量、55 种枚举和 19 种结构。

1) 类

GDI+ API 中共有 54 个类，核心类是 `Graphics`，它是实际绘制直线、曲线、图形、图像和文本的类。许多其它 GDI+类是与 `Graphics` 类一起使用的。例如，`DrawLine` 方法接收 `Pen` 对象，该对象中存有所要绘制的线条的属性（颜色、宽度、虚线线型等）。`FillRectangle` 方法可以接收指向 `LinearGradientBrush` 对象的指针，该对象与 `Graphics` 对象配合工作来用一种渐变色填充矩形。`Font` 和 `StringFormat` 对象影响 `Graphics` 对象绘制文本的方式。`Matrix` 对象存储并操作 `Graphics` 对象的仿射变换——旋转、缩放和翻转图像。

GDI+还提供了用于组织图形数据的几种结构类（例如 `Rect`、`Point` 和 `Size`）。而且，某些类的主要作用是结构化数据类型。例如，`BitmapData` 类是 `Bitmap` 类的帮助器，`PathData` 类是 `GraphicsPath` 类的帮助器。

下面是所有 GDI+的 API 类的列表：

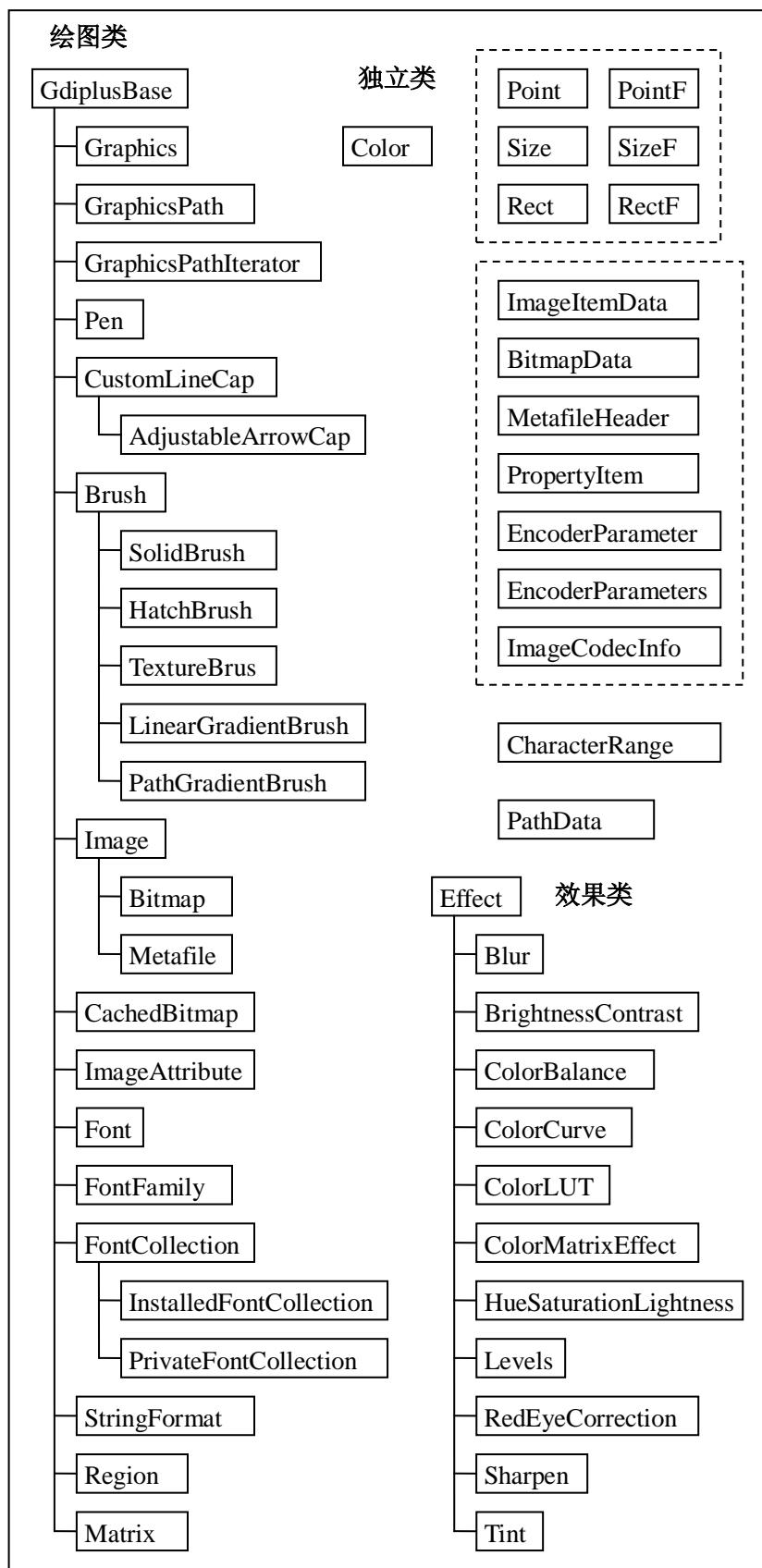
GDI+的 API 类 (54 个)

名称	类	功能
调整箭头帽	<code>AdjustableArrowCap</code>	创建自定义箭头线帽
位图	<code>Bitmap</code>	提供装入和保存矢量和光栅图像的方法，并可以创建和操作光栅图像
位图数据	<code>BitmapData</code>	保存位图的属性
模糊	<code>Blur</code>	将高斯模糊效果作用到图像
亮度对比度	<code>BrightnessContrast</code>	改变图像的亮度和对比度
刷	<code>Brush</code>	定义刷对象
缓存图像	<code>CachedBitmap</code>	用为特点设备显示而优化过的格式存储位图
字符范围	<code>CharacterRange</code>	指定串内字符位置的范围
颜色	<code>Color</code>	保存表示颜色的 32 位值
色平衡	<code>ColorBalance</code>	改变位图的颜色平衡
颜色曲线	<code>ColorCurve</code>	可调整位图的曝光度、密度、对比度、加亮、阴影、色调、白饱和和黑饱和。
颜色查找表	<code>ColorLUT</code>	用于定制位图的颜色调整
颜色矩阵效果	<code>ColorMatrixEffect</code>	对位图进行仿射变换
定制线帽	<code>CustomLineCap</code>	封装了自定义线帽
效果	<code>Effect</code>	作用于图像的效果和调整类的基类
编码器参数	<code>EncoderParameter</code>	保存图像编码器的参数
编码器参数组	<code>EncoderParameters</code>	图像编码器参数的数组

字体	Font	封装了字体的族系、高度、大小和风格等特性
字体集	FontCollection	包含枚举字体集中的字体族系的方法
字体族	FontFamily	封装了构成一个字体族的字体集合
GDI+基	GdiplusBase	提供对 GDI+对象的存储分配与释放，是其它 GDI+类的基类
图形	Graphics	提供绘制图形、图像和文本的方法，存储显示设备和被画项目的属性
图形路径	GraphicsPath	保存一个供绘图用的直线、曲线和形状序列
图形路径迭代器	GraphicsPathIterator	提供从保存在 GraphicsPath 对象中的路径里选择孤立子集的方法
影线刷	HatchBrush	定义具有影线风格和前景色/背景色的矩形刷
色调饱和度亮度	HueSaturationLightness	改变位图的色调 H、饱和度 S 和亮度 L
图像	Image	提供装入和保存矢量和光栅图像的方法
图像属性	ImageAttributes	含渲染时如何操作位图和图元文件颜色的信息
图像编解码信息	ImageCodecInfo	存储与图像编解码有关的信息
图像项数据	ImageItemData	用于存储和获取自定义图像的元数据
已装入字体集	InstalledFontCollection	定义表示已装入系统中的字体集
级别	Levels	可调整位图的加亮、阴影和色调
线形梯度刷	LinearGradientBrush	定义线性渐变刷
矩阵	Matrix	表示 3×3 的仿射变换矩阵
图元文件	Metafile	定义包含描述一系列图形 API 调用记录的图形元文件，可被记录（构造）和回放（显示）
图元文件头	MetafileHeader	保存关联图元文件的性质
路径数据	PathData	GraphicsPath 和 GraphicsPathIterator 类的助手类，用于获取和设置路径中的数据点及其类型
路径梯度刷	PathGradientBrush	保存颜色的梯度属性，用于渐变色填充路径内部
笔	Pen	用于绘制直线和曲线的笔对象
点	Point	封装 2D 整数坐标系统中的点
浮点点	PointF	封装 2D 浮点坐标系统中的点
专用字体集	PrivateFontCollection	保存用于特定应用程序的字体集，可含未装入系统中的字体
特性项	PropertyItem	Image 和 Bitmap 类的助手类，保存一块 (piece) 图像元数据
矩形	Rect	保存矩形的左上角、宽度和高度之对象（整数）
浮点矩形	RectF	保存矩形的左上角、宽度和高度之对象（浮点数）
红眼校正	RedEyeCorrection	校正有时在闪光照片中出现的红眼
区域	Region	描述显示表面的范围，可以是任意形状
锐化	Sharpen	调整位图的清晰度
大小	Size	封装 2D 整数坐标系统中的宽和高
浮点大小	SizeF	封装 2D 浮点数坐标系统中的宽和高
实心刷	SolidBrush	定义实心颜色的刷子对象
串格式	StringFormat	封装文本的格式（layout）信息和显示操作
纹理刷	TextureBrush	用于填充的包含图像对象的刷子

浓淡	Tint	改变位图的色彩浓淡
----	------	-----------

下面是 GDI+ API 类的层次结构图：



GDI+类的层次结构图

2) 函数

GDI+命名空间中的函数（12个）

名称	函数	功能
关闭 GDI+	GdiplusShutdown	清除 GDI+所使用的资源
启动 GDI+	GdiplusStartup	初始化 GDI+
获取图像解码器	GetImageDecoders	获取含有可用图像解码器信息的 ImageCodecInfo 对象数组
获取图像解码器的大小	GetImageDecodersSize	获取含有可用图像解码器的数目
获取图像编码器	GetImageEncoders	获取含有可用图像编码器信息的 ImageCodecInfo 对象数组
获取图像编码器的大小	GetImageEncodersSize	获取含有可用图像编码器的数目
获取像素格式大小	GetPixelFormatSize	返回指定像素格式的每像素二进制位数
是否为 α 像素格式	IsAlphaPixelFormat	确定指定像素格式是否有 α 分量
是否为规范像素格式	IsCanonicalPixelFormat	确定指定像素格式是否为规范格式之一
是否为扩展像素格式	IsExtendedPixelFormat	确定指定像素格式是否使用 16 位色
是否为索引像素格式	IsIndexedPixelFormat	确定指定像素格式是否是索引格式
对象类型是否有效	ObjectTypeIsValid	确定 ObjectType 枚举元素是否表示一个有效对象类型

3) 常量

GDI+中定义了如下 6 类图像常量（226个）：(GdiplusImaging.h)

类型	常量	说明
图像文件格式 (11个)	ImageFormatBMP	BMP (BitMaP 位图)
	ImageFormatEMF	EMF (Enhanced MetaFile 增强图元文件)
	ImageFormatEXIF	Exif (Exchangeable Image File 可交换图像文件)
	ImageFormatGIF	GIF (Graphics Interchange Format 图形交换格式)
	ImageFormatIcon	Icon (图标)
	ImageFormatJPEG	JPEG (Joint Photographic Experts Group 联合图象专家组)
	ImageFormatMemoryBMP	从内存位图构造的图像
	ImageFormatPNG	PNG (Portable Network Graphics 可移植网络图形)
	ImageFormatTIFF	TIFF (Tagged Image File Format 标签图像文件格式)
	ImageFormatUndefined	不能确定格式
图像帧维 (13个)	ImageFormatWMF	WMF (Windows Metafile Format 视窗图元文件格式)
	FrameDimensionPage	多帧 TIFF 图像
	FrameDimensionTime	多帧 GIF 图像
图像编码器 (13个)	EncoderChrominanceTable	色度表
	EncoderColorDepth	颜色深度
	EncoderColorSpace	颜色空间

	EncoderCompression	压缩
	EncoderLuminanceTable	亮度表
	EncoderQuality	质量
	EncoderRenderMethod	渲染方法
	EncoderSaveFlag	保存标志
	EncoderScanMethod	扫描方法
	EncoderTransformation	变换
	EncoderVersion	版本
	EncoderImageItems	图像项
	EncoderSaveAsCMYK	保存为 CMYK (Cyan 青、Magenta 品红、Yellow 黄、black 黑, 用于印刷的四分色)
图像像素格式(14个)	PixelFormat1bppIndexed	每像素 1 位, 索引色
	PixelFormat4bppIndexed	每像素 4 位, 索引色
	PixelFormat8bppIndexed	每像素 8 位, 索引色
	PixelFormat16bppARGB1555	每像素 16 位, α 分量 1 位、RGB 分量各 5 位
	PixelFormat16bppGrayScale	每像素 16 位, 灰度
	PixelFormat16bppRGB555	每像素 16 位, RGB 分量各 5 位, 另 1 位未用
	PixelFormat16bppRGB565	每像素 16 位, RB 分量各 5 位、G 分量 6 位
	PixelFormat24bppRGB	每像素 24 位, RGB 分量各 8 位
	PixelFormat32bppARGB	每像素 32 位, α RGB 分量各 8 位
	PixelFormat32bppPARGB	每像素 32 位, α RGB 分量各 8 位, RGB 分量预乘 α 分量
	PixelFormat32bppRGB	每像素 24 位, RGB 分量各 8 位, 另 8 位未用
	PixelFormat48bppRGB	每像素 48 位, RGB 分量各 16 位
	PixelFormat64bppARGB	每像素 64 位, α RGB 分量各 16 位
	PixelFormat64bppPARGB	每像素 64 位, α RGB 分量各 16 位, RGB 分量预乘 α 分量
图像特性标志类型(9个)	PixelFormat4bppIndexed	格式为每像素 4 位, 索引色
	PropertyTagTypeASCII	值数据成员为以 null 结尾的 ASCII 字符串
	PropertyTagTypeByte	值数据成员为字节数组
	PropertyTagTypeLong	值数据成员为 32 位无符号长整数的数组
	PropertyTagTypeRational	值数据成员为 32 位无符号长整数对的数组, 每对数中的第一个整数为分子, 第二个整数为分母
	PropertyTagTypeShort	值数据成员为 16 位无符号短整数的数组
	PropertyTagTypeSLONG	值数据成员为 32 位有符号长整数的数组
	PropertyTagTypeSRational	值数据成员为 32 位有符号长整数对的数组, 每对数中的第一个整数为分子, 第二个整数为分母
	PropertyTagTypeUndefined	值数据成员为字节数组, 可保存任何数据类型的值
图像特性标志(217个)	PropertyTagGpsVer ~	GPS (Global Positioning Systems 全球定位系统) 版本
	PropertyTagGpsDestDist	(0x0000) ~ 到目标点的距离 (0x001A) (27 个)
	PropertyTagNewSubfileType ~	子文件数据类型 (0x00FE) ~
	PropertyTagPageNumber	被扫描图像的页数 (0x0129) (44 个)
	PropertyTagTransferFunction	图像传送函数表 (0x012D)

PropertyTagSoftwareUsed	指定用于生成图像的设备之软件或固件的名称和版本的以 null 结尾的字符串 (0x0131)
PropertyTagDateTime	图像创建的日期和时间 (0x0132)
PropertyTagArtist ~	指定图像创建者姓名的以 null 结尾的字符串(0x013B)
PropertyTagTileByteCounts	~ 标题的字节数 (0x0145) (11 个)
PropertyTagInkSet ~	在分开图像中使用的墨水集 (0x014C)
PropertyTagNumberOfInks	~ 墨水数目 (0x014D) (3 个)
PropertyTagDotRange ~	对应于 0% 点和 100% 点的颜色分量值 (0x0150) ~
PropertyTagTransferRange	扩充传送函数范围的值表 (0x0156) (7 个)
PropertyTagJPEGProc ~	JPEG 压缩过程 (0x0200) ~
PropertyTagImageTitle	图像标题的以 null 结尾的字符串 (0x0320) (17 个)
PropertyTagResolutionXUnit ~	显示水平分辨率的单位 (0x5001) ~ (27 个)
PropertyTagThumbnailData	RGB 或 JPEG 中的原始缩略图中的位数据 (0x501B)
PropertyTagThumbnailImageWidth ~	略图像的每行像素数 (0x5020) ~ (28 个)
PropertyTagThumbnailCopyRight	含缩略图像版权信息的以 null 结尾的字符串(0x503B)
PropertyTagLuminanceTable	亮度表 (0x5090)
PropertyTagFrameDelay ~	GIF 动画中两帧之间的延时, 单位为 10 毫秒(0x5100)
PropertyTagPaletteHistogram	~ 调色板直方图 (0x5113) (9 个)
PropertyTagCopyright ~	含版权信息的以 null 结尾的字符串 (0x8298B) ~
PropertyTagExifCfaPattern	颜色滤波器数组 (0xA302) (48 个)

4) 枚举

GDI+定义了 55 种枚举, 它们都是相关常数的集合。例如, LineJoin 枚举包含元素 Bevel、Miter 和 Round, 它们指定可用于连接两个线条的线型。下面是所有枚举类型的列表:

GDI+枚举类型 (55 种)

枚举类型	名称	枚举类型	名称
BrushType	刷类型	ImageType	图像类型
ColorAdjustType	颜色调整类型	InterpolationMode	插值类型
ColorChannelFlags	颜色通道标志	ItemDataPosition	项数据位置
ColorMatrixFlags	颜色矩阵标志	LinearGradientMode	线性梯度模式
CombineMode	组合模式	LineCap	线帽
CompositingMode	合成模式	LineJoin	线连接
CompositingQuality	合成质量	MatrixOrder	矩阵序(左右乘)
CoordinateSpace	坐标空间	MetafileFrameUnit	图元文件帧单位
CurveAdjustments	曲线调整	MetafileType	图元文件类型
CurveChannel	曲线通道	ObjectType	对象类型
DashCap	虚线帽	PaletteFlags	调色板标志
DashStyle	虚线风格	PaletteType	调色板类型

DitherType	抖动类型		PathPointType	路径点类型
DriverStringOptions	驱动器串选项		PenAlignment	笔对齐
EmfPlusRecordType	EMF+等图元文件记录类型		PenType	笔类型
EmfToWmfBitsFlags	EMF 转 WMF 的标志位		PixelOffsetMode	像素偏移模式
EmfType	EMF 类型		RotateFlipType	旋转翻转类型
EncoderParameterValueType	编码器参数值类型		SmoothingMode	平滑模式
EncoderValue	编码器值		Status	状态
FillMode	填充模式		StringAlignment	串对齐
FlushIntention	刷新意图		StringDigitSubstitute	串数字替换
FontStyle	字体风格		StringFormatFlags	串格式标志
HatchStyle	影线风格		StringTrimming	串修整
HistogramFormat	直方图格式		TextRenderingHint	文本渲染提示
HotkeyPrefix	热键前缀		Unit	单位
ImageCodecFlags	图像编解码标志		WarpMode	弯曲模式
ImageFlags	图像标志		WrapMode	覆盖模式
ImageLockMode	图像加锁模式			

5) 结构

GDI+ API 中还定义了 19 种结构，用于 GDI+的各种函数调用中。下面是所有 GDI+ API 结构的列表：

GD+ API 中的结构（19 种）

结构	名称
BlurParams	模糊参数
BrightnessContrastParams	亮度对比度参数
ColorBalanceParams	颜色平衡参数
ColorCurveParams	颜色曲线参数
ColorLUTParams	颜色查找表参数
ColorMap	颜色映射
ColorMatrix	颜色矩阵
ColorPalette	颜色调色板
ENHMETAHEADER3	增强图元文件头
GdiplusAbort	GDI+异常中断
GdiplusStartupInput	GDI+启动输入
GdiplusStartupOutput	GDI+启动输出
HueSaturationLightnessParams	色调饱和度亮度参数
LevelsParams	级别参数
PWMFRect16	可定位 WMF 矩形 (INT16 整数值)
RedEyeCorrectionParams	红眼校正参数
SharpenParams	锐化参数

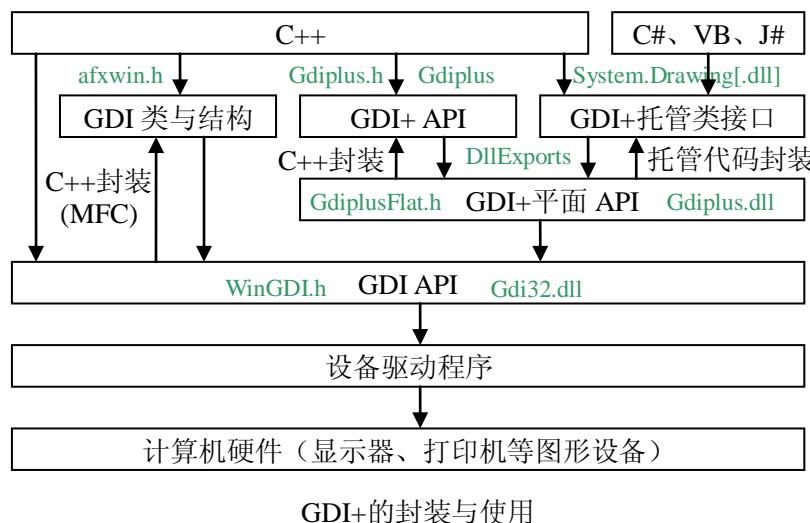
TintParams	浓淡参数
WmfPlaceableFileHeader	可定位 WMF 文件头

6) GDI+平面 API

GDI+暴露出(exposes)一个平面(flat)API, 它包含大约 600 个函数, 被实现在 Gdiplus.dll 中, 声明在 Gdiplusflat.h 内。这些函数被包装到了前面讨论过的 GDI+ API 的 54 个 C++类的集合之中。不要直接调用这些函数, 而推荐用调用类成员方法来替代。因为微软产品支持服务 (Microsoft Product Support Services), 不会为直接调用平面 API 的代码提供支持。

作为 C++封装的替代方案, 微软网络框架 (Microsoft .NET Framework), 提供了 GDI+ 的一个托管代码封装类集, 包含大约 60 个类、50 个枚举和 8 个结构。它们属于下列命名空间:

```
System.Drawing
System.Drawing.Drawing2D
System.Drawing.Imaging
System.Drawing.Text
System.Drawing.Printing
```



GDI+的封装与使用

这两种包装 (C++和托管代码) 都采用了面向对象方法, 所以二者在将参数传递给封装的方法和将参数传递给平面 API 函数的方式上存在差别。

例如, C++的 Matrix 类对象中, 有一个 nativeMatrix 字段 (field), 它指向一个 GpMatrix 类型的内部变量。当你传递一些参数到一个 Matrix 对象的方法时, 该方法会将这些参数向下传递到平面 API 函数, 但是该方法还将 nativeMatrix 字段也作为输入参数传递给了平面 API 函数。例如:

```
Status Shear(REAL shearX, REAL shearY, // GDI+ API 中 Matrix 类的方法
             MatrixOrder order = MatrixOrderPrepend) {
    .....
    GdipShearMatrix(nativeMatrix, shearX, shearY, order); // 平面 API 函数
    .....
}
```

其中，平面 API 函数 GdipShearMatrix 的函数原型为：

```
GpStatus WINGDIPAPI GdipShearMatrix(GpMatrix *matrix, REAL shearX, REAL  
shearY, GpMatrixOrder order)
```

Matrix 构造函数传递一个 GpMatrix 指针变量（输出参数）的地址到平面 API 函数：

```
GpStatus WINGDIPAPI GdipCreateMatrix(GpMatrix **matrix);
```

该函数创建并初始化一个内部 GpMatrix 变量，然后将 GpMatrix 的地址，赋值给指针变量。接着构造函数将该指针的值复制到 nativeMatrix 字段。如：

```
Matrix() {  
    GpMatrix *matrix = NULL;  
    lastResult = DllExports::GdipCreateMatrix(&matrix);  
    SetNativeMatrix(matrix);  
}  
VOID SetNativeMatrix(GpMatrix *nativeMatrix) {  
    this->nativeMatrix = nativeMatrix;  
}
```

而封装类中的克隆方法 Clone，不接受参数，而经常是传递两个参数到底层的 GDI+平面 API 函数。Matrix::Clone 传递 nativeMatrix（作为输入参数）和 GpMatrix 指针变量的地址（作为输出参数）给平面 API 的 GdipCloneMatrix 函数：

```
GpStatus WINGDIPAPI GdipCloneMatrix(GpMatrix *matrix, GpMatrix **cloneMatrix);
```

例如：

```
Matrix *Clone() const {  
    GpMatrix *cloneMatrix = NULL;  
    .....  
    GdipCloneMatrix(nativeMatrix, &cloneMatrix));  
    .....  
    return new Matrix(cloneMatrix);  
}
```

该平面 API 函数返回一个 GpStatus 类型的值。在 GdiplusGpStubs.h 中，枚举类型 GpStatus 被定义为与枚举类型 Status 等价：

```
typedef Status GpStatus;
```

在封装类中，大多数方法都返回一个状态值，指出方法是否成功。但是也有一些方法用布尔变量来返回状态。例如，Matrix 类的 IsInvertible 方法：

```
BOOL IsInvertible() const {  
    BOOL result = FALSE;  
    .....  
    GdipIsMatrixInvertible(nativeMatrix, &result);  
    return result;  
}
```

其中，平面 API 函数 GdipIsMatrixInvertible 的函数原型为：

```
GpStatus WINGDIPAPI GdipIsMatrixInvertible(GDIPCONST GpMatrix *matrix,  
BOOL *result);
```

另一个封装类是 Color，它只有一个（被定义为 DWORD 的）ARGB 类型的字段。当你传递一个 Color 对象到某个封装方法时，该方法也会将 ARGB 字段传到底层的 GDI+平面 API 函数。例如 Pen::SetColor：

```
Status SetColor(const Color& color) {  
    .....  
    GdipSetPenColor(nativePen, color.GetValue());  
}
```

Color::GetValue 方法返回 ARGB 字段的值。其中，平面 API 函数 GdipSetPenColor 的函数原型为：

```
GpStatus WINGDIPAPI GdipSetPenColor(GpPen *pen, ARGB argb);
```

6.2 GDI+的 MFC 编程

1. 基础

封装在 GDI+ API 中的各种 C++类、函数、常量、枚举和结构，都被定义在 Gdiplus.h 头文件所包含的一系列头文件中。所以，采用 MFC 进行 GDI+编程，必须包含 Gdiplus.h 头文件。

由上节最后一小小节“5. 6) GDI+平面 API”中的讨论可知，封装在 GDI+类中方法，最后都需要调用 GDI+平面 API 中的相关底层函数，才能完成实际的操作。所以，为了运行 GDI+应用程序，在操作系统平台中，必须安装动态链接库 Gdiplus.dll。

该动态链接库所对应的静态库文件为 GdiPlus.lib，而且它不是 C++ 和 MFC 的缺省链接库。所以，必须在项目设置，添加该库作为链接器输入的附加依赖项。

因为在 Gdiplus.h 头文件中，将所有的 GDI+的类、函数、常量、枚举和结构等都定义在了命名空间 Gdiplus 中。所以，一般在 GDI+程序中，都必须使用如下的命名空间声明：

```
using namespace Gdiplus;
```

例如：

```
#include <gdiplus.h>
using namespace Gdiplus;
....
```

1) GdiPlus.h

```
/****************************************************************************
 * Copyright (c) 1998-2001, Microsoft Corp. All Rights Reserved.
 * Module Name:
 *     Gdiplus.h
 * Abstract:
 *     GDI+ public header file
 */
#ifndef _GDIPLUS_H
#define _GDIPLUS_H
struct IDirectDrawSurface7;
typedef signed short INT16;
typedef unsigned short UINT16;
#include <pshpack8.h> // set structure packing to 8
namespace Gdiplus
{
    namespace DllExports {
        #include "GdiplusMem.h"
    };
    #include "GdiplusBase.h"
```

```

#include "GdiplusEnums.h"
#include "GdiplusTypes.h"
#include "GdiplusInit.h"
#include "GdiplusPixelFormats.h"
#include "GdiplusColor.h"
#include "GdiplusMetaHeader.h"
#include "GdiplusImaging.h"
#include "GdiplusColorMatrix.h"
#include "GdiplusGpStubs.h"
#include "GdiplusHeaders.h"
namespace DllExports {
    #include "GdiplusFlat.h"
};

#include "GdiplusImageAttributes.h"
#include "GdiplusMatrix.h"
#include "GdiplusBrush.h"
#include "GdiplusPen.h"
#include "GdiplusStringFormat.h"
#include "GdiplusPath.h"
#include "GdiplusLineCaps.h"
#include "GdiplusMetafile.h"
#include "GdiplusGraphics.h"
#include "GdiplusCachedBitmap.h"
#include "GdiplusRegion.h"
#include "GdiplusFontCollection.h"
#include "GdiplusFontFamily.h"
#include "GdiplusFont.h"
#include "GdiplusBitmap.h"
#include "GdiplusImageCodec.h"
}; // namespace Gdiplus
#include <poppack.h>      // pop structure packing back to previous state
#endif // !_GDIPLUS_HPP

```

2) GDI+的初始化与清除

为了在 MFC 应用程序中使用采用 C++封装的 GDI+ API，必须在 MFC 项目的应用程序类中，调用 GDI+ 命名空间中的 GDI+ 启动函数 GdiplusStartup 和 GDI+ 关闭函数 GdiplusShutdown，来对 GDI+进行初始化（装入动态链接库 Gdiplus.dll，或锁定标志+1）和清除（卸载动态链接库 Gdiplus.dll，或锁定标志-1）工作。它们一般分别在应用程序类的 InitInstance 和 ExitInstance 重载成员函数中调用。

函数 GdiplusStartup 和 GdiplusShutdown，都被定义在 GdiplusInit.h 头文件中：

```

Status WINAPI GdiplusStartup(
    OUT ULONG_PTR *token,
    const GdiplusStartupInput *input,

```

```

    OUT GdiplusStartupOutput *output);
void GdiplusShutdown(ULONG_PTR token);

```

其中：

- 类型 ULONG_PTR，是用无符号长整数表示的指针，被定义在 basetsd.h 头文件中：

```
typedef _W64 unsigned long ULONG_PTR;
```

输出参数 token（权标），供关闭 GDI+的函数使用，所以必须设置为应用程序类的成员变量（或全局变量，不提倡）。

- 结构 GdiplusStartupInput 和 GdiplusStartupOutput，也都被定义在 GdiplusInit.h 头文件中：

```

struct GdiplusStartupInput {
    UINT32 GdiplusVersion;           // Must be 1
    DebugEventProc DebugEventCallback; // Ignored on free builds
    BOOL SuppressBackgroundThread;   // FALSE unless you're prepared to call
                                    // the hook/unhook functions properly
    BOOL SuppressExternalCodecs;     // FALSE unless you want GDI+ only to
use
                                    // its internal image codecs.

GdiplusStartupInput(
    DebugEventProc debugEventCallback = NULL,
    BOOL suppressBackgroundThread = FALSE,
    BOOL suppressExternalCodecs = FALSE) {
    GdiplusVersion = 1;
    DebugEventCallback = debugEventCallback;
    SuppressBackgroundThread = suppressBackgroundThread;
    SuppressExternalCodecs = suppressExternalCodecs;
}
};

struct GdiplusStartupOutput {
    NotificationHookProc NotificationHook;
    NotificationUnhookProc NotificationUnhook;
};

■ GDI+启动输入结构指针参数 input，一般取缺省构造值即可，即（设：无调试事件回调过程、不抑制背景线程、不抑制外部编解码）：
    input = GdiplusStartupInput(NULL, FALSE, FALSE);

■ GDI+启动输出结构指针参数 output，一般不需要，取为 NULL 即可。

```

注意，采用 MFC 进行 GDI+ API 编程时，在使用任何 GDI+的功能调用之前，必须先调用 GDI+启动函数 GdiplusStartup 来进行初始化 GDI+的工作；在完成所有的 GDI+功能调用之后，必须调用 GDI+关闭函数 GdiplusShutdown 来进行清除 GDI+的工作。

例如：（创建一个名为 GdipDraw 的 MFC 单文档应用程序项目，在项目属性中添加 GdiPlus.lib 库作为链接器输入的附加依赖项）

```

// GdipDraw.h
.....
class CGdipDrawApp : public CWinApp

```

```

{
public:
    CGdipDrawApp();
    ULONG_PTR m_gdiplusToken;
    .....
};

// GdipDraw.cpp
.....
#include <gdiplus.h>
using namespace Gdiplus;
.....
BOOL CGdipDrawApp::InitInstance()
{
    // 如果一个运行在 Windows XP 上的应用程序清单指定要
    // 使用 ComCtl32.dll 版本 6 或更高版本来启用可视化方式,
    // 则需要 InitCommonControlsEx()。否则, 将无法创建窗口。
    INITCOMMONCONTROLSEX InitCtrls;
    InitCtrls.dwSize = sizeof(InitCtrls);
    // 将它设置为包括所有要在应用程序中使用的
    // 公共控件类。
    InitCtrls.dwICC = ICC_WIN95_CLASSES;
    InitCommonControlsEx(&InitCtrls);

    /*注意：下面这两个语句必须加在 CWinApp:: InitInstance ();语句之前，不然以后会造成视图窗口不能自动重画、程序中不能使用字体等等一系列问题。*/
    GdiplusStartupInput gdiplusStartupInput;
    GdiplusStartup(&m_gdiplusToken, &gdiplusStartupInput, NULL);

    CWinApp::InitInstance();
    .....
}

int CGdipDrawApp::ExitInstance() // 该函数是自己利用属性窗口, 添加的重写函数
{
    // TODO: 在此添加专用代码和/或调用基类
    GdiplusShutdown(m_gdiplusToken);
    return CWinApp::ExitInstance();
}

```

3) 几何辅助类

在 GDI+ API 中, 定义了许多绘图的辅助类, 常用的有点、大小和矩形等几何类。它们都是没有基类的独立类, 被定义在头文件 GdiplusTypes.h 中。下面逐个进行介绍:

(1) Point[F] (点)

GDI+中，有两种类型的点：整数点（对应于 Point 类）和浮点数点（对应于 PointF 类）。下面分别加以介绍：

- 整数点类 Point:

```
class Point {  
public:  
    Point() {X = Y = 0;}  
    Point(const Point &point) {X = point.X; Y = point.Y;}  
    Point(const Size &size) {X = size.Width; Y = size.Height;}  
    Point(INT x, INT y) {X = x; Y = y;}  
    Point operator+(const Point& point) const {return Point(X + point.X, Y + point.Y);}  
    Point operator-(const Point& point) const {return Point(X - point.X, Y - point.Y);}  
    BOOL Equals(const Point& point) {return (X == point.X) && (Y == point.Y);}  
public:  
    INT X; // 大写 X、Y  
    INT Y;  
};
```

其中：

```
typedef int INT; // 4 字节有符号整数 (windef.h)
```

注意，这里的点与 GDI 的区别：

```
POINT 和 CPoint: {x; y;} // 小写 x、y
```

- 浮点数点类 PointF:

```
class PointF {  
public:  
    PointF() {X = Y = 0.0f;}  
    PointF(const PointF &point) {X = point.X; Y = point.Y;}  
    PointF(const SizeF &size) {X = size.Width; Y = size.Height;}  
    PointF(REAL x, REAL y) {X = x; Y = y;}  
    PointF operator+(const PointF& point) const {return PointF(X + point.X, Y +  
    point.Y);}  
    PointF operator-(const PointF& point) const {  
        return PointF(X - point.X, Y - point.Y);  
    }  
    BOOL Equals(const PointF& point) {  
        return (X == point.X) && (Y == point.Y);  
    }  
public:  
    REAL X;  
    REAL Y;  
};
```

其中：

```
typedef float REAL; // 4 字节浮点数 (GdiplusTypes.h)
```

注意，浮点数版的几何对象和绘图函数，是 GDI+新增的功能，这些在各种工程技术领域都非常有用。因为一般的实际图形设计，都是基于实数坐标的。包括机械（机床/汽车/轮船/飞机等）、建筑（房屋/桥梁/道路/堤坝等）和图形动画设计（形状/物体/人物/背景/轨迹等）等设计，都必须使用浮点参数和坐标系。

(2) Size[F] (大小)

GDI+中，也有两种类型的大小（尺寸）：整数大小（对应于 Size 类）和浮点数大小（对应于 SizeF 类）。下面分别加以介绍：

- 整数大小类 Size：

```
class Size {
public:
    Size() {Width = Height = 0;}
    Size(const Size& size) {Width = size.Width; Height = size.Height;}
    Size(INT width, INT height) {Width = width; Height = height;}
    Size operator+(const Size& sz) const {
        return Size(Width + sz.Width, Height + sz.Height);
    }
    Size operator-(const Size& sz) const {
        return Size(Width - sz.Width, Height - sz.Height);
    }
    BOOL Equals(const Size& sz) const {
        return (Width == sz.Width) && (Height == sz.Height);
    }
    BOOL Empty() const {return (Width == 0 && Height == 0);}
public:
    INT Width; // 不是 cx 和 cy
    INT Height;
};
```

注意，这里的大小与 GDI 的区别：

SIZE 和 CSize: {**cx**; **cy**;} // 不是宽和高

- 浮点数大小类 SizeF：

```
class SizeF {
public:
    SizeF() {Width = Height = 0.0f;}
    SizeF(const SizeF& size) {Width = size.Width; Height = size.Height;}
    SizeF(REAL width, REAL height) {Width = width; Height = height;}
    SizeF operator+(const SizeF& sz) const {
        return SizeF(Width + sz.Width, Height + sz.Height);
    }
    SizeF operator-(const SizeF& sz) const {
        return SizeF(Width - sz.Width, Height - sz.Height);
    }
};
```

```

    }
    BOOL Equals(const SizeF& sz) const {
        return (Width == sz.Width) && (Height == sz.Height);
    }
    BOOL Empty() const {return (Width == 0.0f && Height == 0.0f);}
public:
    REAL Width;
    REAL Height;
};

```

(3) Rect[F] (矩形)

GDI+中，也有两种类型的矩形：整数矩形（对应于 Rect 类）和浮点数矩形（对应于 RectF 类）。下面分别加以介绍：

- 整数矩形类 Rect：

```

class Rect {
public:
    Rect() {X = Y = Width = Height = 0;}
    Rect(INT x, INT y, INT width, INT height);
    Rect(const Point& location, const Size& size);
    Rect* Clone() const;
    VOID GetLocation(OUT Point* point) const;
    VOID GetSize(OUT Size* size) const;
    VOID GetBounds(OUT Rect* rect) const;
    INT GetLeft() const {return X;}
    INT GetTop() const {return Y;}
    INT GetRight() const {return X+Width;}
    INT GetBottom() const {return Y+Height;}
    BOOL IsEmptyArea() const{return (Width <= 0) || (Height <= 0);}
    BOOL Equals(const Rect & rect) const;
    BOOL Contains(INT x, INT y) const;
    BOOL Contains(const Point& pt) const;
    BOOL Contains(Rect& rect) const;
    VOID Inflate(INT dx, INT dy) {
        X -= dx;           Y -= dy;
        Width += 2*dx;     Height += 2*dy;
    }
    VOID Inflate(const Point& point) {Inflate(point.X, point.Y);}
    BOOL Intersect(const Rect& rect) {return Intersect(*this, *this, rect);}
    static BOOL Intersect(OUT Rect& c, const Rect& a, const Rect& b);
    BOOL IntersectsWith(const Rect& rect) const;
    static BOOL Union(OUT Rect& c, const Rect& a, const Rect& b);
    VOID Offset(const Point& point);
    VOID Offset(INT dx, INT dy);

```

```

public:
    INT X; // 不是 left 和 top
    INT Y;
    INT Width; // 更不是 right 和 bottom
    INT Height;
};

```

注意，这里的矩形与 GDI 的区别：

```
RECT: {left; top; right; bottom;} // 不是宽和高
```

虽然 Rect 中的(X, Y)等价于 RECT 的(left, top)，但是 Rect 中的(Width, Height)却不同于 RECT 的(right, bottom)。

- 浮点数矩形类 RectF:

```

class RectF {
public:
    RectF() {X = Y = Width = Height = 0.0f;}
    RectF(REAL x, REAL y, REAL width, REAL height);
    RectF(const PointF& location, const SizeF& size);
    RectF* Clone() const;
    VOID GetLocation(OUT PointF* point) const;
    VOID GetSize(OUT SizeF* size) const;
    VOID GetBounds(OUT RectF* rect) const;
    REAL GetLeft() const;
    REAL GetTop() const;
    REAL GetRight() const;
    REAL GetBottom() const;
    BOOL IsEmptyArea() const;
    BOOL Equals(const RectF & rect) const;
    BOOL Contains(REAL x, REAL y) const;
    BOOL Contains(const PointF& pt) const;
    BOOL Contains(const RectF& rect) const;
    VOID Inflate(REAL dx, REAL dy);
    VOID Inflate(const PointF& point);
    BOOL Intersect(const RectF& rect);
    static BOOL Intersect(OUT RectF& c, const RectF& a, const RectF& b);
    BOOL IntersectsWith(const RectF& rect) const;
    static BOOL Union(OUT RectF& c, const RectF& a, const RectF& b);
    VOID Offset(const PointF& point);
    VOID Offset(REAL dx, REAL dy);
public:
    REAL X;
    REAL Y;
    REAL Width;
    REAL Height;
};

```

在 GDI 的 MFC 封装中，也有点、大小和矩形的

- 结构: (windef.h)

```
typedef struct tagPOINT {LONG x; LONG y;} POINT; // typedef long LONG;
typedef struct tagSIZE {LONG cx; LONG cy;} SIZE;
typedef struct tagRECT {LONG left; LONG top; LONG right; LONG bottom;} RECT;
```

- 类: (atltypes.h)

```
class CPoint : public tagPOINT {
public:
    CPoint() throw();
    CPoint(int initX, int initY) throw();
    .....
}

class CSize : public tagSIZE {
public:
    CSize() throw();
    CSize(int initCX, int initCY) throw();
    .....
}

class CRect : public tagRECT {
public:
    CRect() throw();
    CRect(int l, int t, int r, int b) throw();
    .....
}
```

可见，这几个类都是从对应的结构派生的。GDI 中，之所以有了类还要保留结构，主要是考虑效率和与底层 GDI API 的兼容。

比较可知，GDI 和 GDI+都有对应的类，不过 GDI+没有对应的结构（但有浮点数版类），而 GDI 则没有对应的浮点数版类（但却有结构）。

2. 绘图

与 GDI 相比，GDI+的绘图新增了许多功能。本小节介绍绘图的基本内容，包括颜色、图形、笔、刷、路径、区域等类的功能和使用。

1) Color (颜色)

GDI+中的颜色，与 GDI 中的颜色的最大不同，是增加了一个字节（8 位）的透明分量 alpha (α)，用来表示颜色的透明度：0 透明（看不见前景色，只有背景色）~255 不透明（看不见背景色，只有前景色）。背景色指原有的颜色，前景色指将要绘图的颜色。

(1) 表示

因此，GDI+中的颜色一般都是用四个字节表示：

整数序：(高位→低位)

α (透明)	R(红)	G(绿)	B(蓝)
---------------	------	------	------

字节序：(低字节→高字节)

B(蓝)	G(绿)	R(红)	α (透明)
------	------	------	---------------

因为，Intel CPU 中，多字节整数的低位在前。

在 GDI+中将颜色数据定义为无符号 4 字节长整数类型 DWORD (双字)：

```
typedef DWORD ARGB; // gdipluspixelformats.h
```

其中：typedef unsigned long DWORD; // windef.h

另外，GDI 中没有专门的颜色类，只有一个颜色类型 COLORREF，也定义为：

```
typedef DWORD COLORREF; // 0x00bbggrr (windef.h)
```

和一个生成颜色的宏：

```
COLORREF RGB(BYTE bRed, BYTE bGreen, BYTE bBlue);
```

其中：typedef unsigned char BYTE; // 单字节无符号字符整数

GDI+的 Color 类，不仅包含了同样的颜色数据（改名为 ARGB），而且还有多个构造函数和其他辅助函数、枚举和常量。

Color 类的构造函数中，最主要的是：

```
Color(BYTE a, BYTE r, BYTE g, BYTE b); // a 为 alpha 分量  $\alpha$ 
```

但也有一个缺省构造函数：

```
Color(VOID); // 不透明黑色，相当于 Color(255, 0, 0, 0);
```

还有一个与 GDI 兼容的构造函数：

```
Color(BYTE r, BYTE g, BYTE b); // 不透明色，相当于 Color(255, r, g, b);
```

你也可以直接用含颜色数据的 4 字节无符号整数，来构造 Color 类的对象：

```
Color(ARGB argb); // 相当于 Color(a, r, g, b);
```

该整数可以由 Color 类的静态函数：

```
static ARGB MakeARGB(BYTE a, BYTE r, BYTE g, BYTE b);
```

或动态函数：

```
ARGB GetValue(VOID);
```

获得。

你也可以用 Color 的函数：

```
COLORREF ToCOLORREF() const;
```

将 Color 对象中的颜色，转换为 GDI 的颜色类型。

Color 类还提供了各个颜色分量的获取函数及其简化版：

```
BYTE GetAlpha() const;    BYTE GetA() const;  
BYTE GetRed() const;     BYTE GetR() const;  
BYTE GetGreen() const;   BYTE GetG() const;  
BYTE GetBlue() const;    BYTE GetB() const;
```

你也可以先用宏：

```
BYTE GetRValue(DWORD rgb); // COLORREF rgb
```

```
BYTE GetGValue(DWORD rgb); // COLORREF rgb
```

```
BYTE GetBValue(DWORD rgb); // COLORREF rgb
```

| 获取 COLORREF 的 R、G、B 值，然后再用这些值调用 Color 类的构造函数来创建 Color

对象。例如：

```
COLORREF crCol = colDlg.GetColor();
BYTE r = GetRValue(crCol), g = GetGValue(crCol), b = GetBValue(crCol);
Color col(r, g, b);
```

(2) 头文件

```
/****************************************************************************
** Copyright (c) 1998-2001, Microsoft Corp. All Rights Reserved.
** Module Name:
**     GdiplusColor.h
** Abstract:
**     GDI+ Color Object
\****************************************************************************/
#ifndef _GDIPLUSCOLOR_H
#define _GDIPLUSCOLOR_H

//-----
// Color mode 颜色模式
//-----

enum ColorMode {
    ColorModeARGB32 = 0,
    ColorModeARGB64 = 1
};

//-----
// Color Channel flags 颜色通道标志
//-----

enum ColorChannelFlags {
    ColorChannelFlagsC = 0,
    ColorChannelFlagsM,
    ColorChannelFlagsY,
    ColorChannelFlagsK,
    ColorChannelFlagsLast
};

//-----
// Color 颜色构造函数
//-----

class Color {
public:
    Color() { Argb = (ARGB)Color::Black; }
    // Construct an opaque Color object with the specified Red, Green, Blue values.
    // Color values are not premultiplied.
    Color(BYTE r, BYTE g, BYTE b) { Argb = MakeARGB(255, r, g, b); }
    Color(BYTE a, BYTE r, BYTE g, BYTE b) { Argb = MakeARGB(a, r, g, b); }
    Color(ARGB argb) { Argb = argb; }
}
```

```

// 获取颜色分量
BYTE GetAlpha() const { return (BYTE)(Argb >> AlphaShift); }
BYTE GetA() const { return GetAlpha(); }
BYTE GetRed() const { return (BYTE)(Argb >> RedShift); }
BYTE GetR() const { return GetRed(); }
BYTE GetGreen() const { return (BYTE)(Argb >> GreenShift); }
BYTE GetG() const { return GetGreen(); }
BYTE GetBlue() const { return (BYTE)(Argb >> BlueShift); }
BYTE GetB() const { return GetBlue(); }

// 获取/设置颜色
ARGB GetValue() const { return Argb; }
VOID SetValue(ARGB argb) { Argb = argb; }
VOID SetFromCOLORREF(COLORREF rgb) {
    Argb = MakeARGB(255, GetRValue(rgb), GetGValue(rgb), GetBValue(rgb));
}
COLORREF ToCOLORREF() const {
    return RGB(GetRed(), GetGreen(), GetBlue());
}

```

(3) 颜色枚举常量

```

public:
// Common color constants 通用颜色常量 (共 141 个)
enum {
    AliceBlue      = 0xFFFF0F8FF, // 艾丽丝蓝
    AntiqueWhite   = 0xFFFAEBD7, // 古董白
    Aqua           = 0xFF00FFFF, // 水绿
    Aquamarine     = 0xFF7FFF4D, // 碧绿
    Azure          = 0xFFFF0FFF, // 天蓝
    Beige          = 0xFFF5F5DC, // 米色
    Bisque         = 0xFFFFE4C4, // 汤黄
    Black          = 0xFF000000, // 黑
    BlanchedAlmond = 0xFFFFEBCD, // 布兰奇杏黄
    Blue           = 0xFF0000FF, // 蓝
    BlueViolet     = 0xFF8A2BE2, // 蓝紫
    Brown          = 0xFFA52A2A, // 棕褐
    BurlyWood      = 0xFFDEB887, // 实木
    CadetBlue      = 0xFF5F9EA0, // 军校蓝
    Chartreuse     = 0xFF7FFF00, // 查特酒绿
    Chocolate      = 0xFFD2691E, // 巧克力色
    Coral          = 0xFFFF7F50, // 珊瑚色
    CornflowerBlue = 0xFF6495ED, // 矢车菊蓝
    Cornsilk        = 0xFFFFF8DC, // 玉米黄
    Crimson        = 0xFFDC143C, // 深红
}

```

Cyan	= 0xFF00FFFF, // 青
DarkBlue	= 0xFF00008B, // 暗蓝
DarkCyan	= 0xFF008B8B, // 暗青
DarkGoldenrod	= 0xFFB8860B, // 暗一枝黄花
DarkGray	= 0xFFA9A9A9, // 暗灰
DarkGreen	= 0xFF006400, // 暗绿
DarkKhaki	= 0xFFBDB76B, // 暗黄褐
DarkMagenta	= 0xFF8B008B, // 暗品红
DarkOliveGreen	= 0xFF556B2F, // 暗橄榄绿
DarkOrange	= 0xFFFF8C00, // 暗橙色
DarkOrchid	= 0xFF9932CC, // 暗淡紫
DarkRed	= 0xFF8B0000, // 暗红
DarkSalmon	= 0FFE9967A, // 暗鲜肉色
DarkSeaGreen	= 0xFF8FBC8B, // 暗海绿
DarkSlateBlue	= 0xFF483D8B, // 暗岩蓝
DarkSlateGray	= 0xFF2F4F4F, // 暗岩灰
DarkTurquoise	= 0xFF00CED1, // 暗青绿
DarkViolet	= 0xFF9400D3, // 暗紫
DeepPink	= 0xFFFF1493, // 深粉红
DeepSkyBlue	= 0xFF00BFFF, // 深天蓝
DimGray	= 0xFF696969, // 淡灰
DodgerBlue	= 0xFF1E90FF, // 传单蓝
Firebrick	= 0xFFB22222, // 耐火砖色
FloralWhite	= 0xFFFFFAF0, // 花卉白
ForestGreen	= 0xFF228B22, // 森林绿
Fuchsia	= 0xFFFF00FF, // 灯笼海棠红
Gainsboro	= 0xFFDCDCDC, // 浅灰色
GhostWhite	= 0xFFF8F8FF, // 幽灵白
Gold	= 0xFFFFD700, // 金黄
Goldenrod	= 0xFFDAA520, // 一枝黄花
Gray	= 0xFF808080, // 灰
Green	= 0xFF008000, // 绿
GreenYellow	= 0xFFADFF2F, // 黄绿
Honeydew	= 0xFFFF0FFF0, // 蜜色
HotPink	= 0xFFFF69B4, // 热粉红
IndianRed	= 0xFFCD5C5C, // 印度红
Indigo	= 0xFF4B0082, // 靛蓝
Ivory	= 0xFFFFFFFF0, // 象牙色
Khaki	= 0xFFFF0E68C, // 卡其黄
Lavender	= 0FFE6E6FA, // 薰衣草紫
LavenderBlush	= 0xFFFFF0F5, // 紫红
LawnGreen	= 0xFF7CFC00, // 草坪绿
LemonChiffon	= 0xFFFFFACD, // 柠檬薄绸
LightBlue	= 0xFFADD8E6, // 亮蓝

LightCoral	= 0xFFFF08080, // 亮珊瑚色
LightCyan	= 0xFFE0FFFF, // 亮青
LightGoldenrodYellow	= 0xFFFFAFAD2, // 亮一枝黄
LightGray	= 0xFFD3D3D3, // 亮灰
LightGreen	= 0xFF90EE90, // 亮绿
LightPink	= 0xFFFFB6C1, // 亮粉红
LightSalmon	= 0xFFFFA07A, // 亮鲜肉色
LightSeaGreen	= 0xFF20B2AA, // 亮海绿
LightSkyBlue	= 0xFF87CEFA, // 亮天蓝
LightSlateGray	= 0xFF778899, // 亮岩灰
LightSteelBlue	= 0xFFB0C4DE, // 亮钢蓝
LightYellow	= 0xFFFFFFF0, // 亮黄
Lime	= 0xFF00FF00, // 浅绿
LimeGreen	= 0xFF32CD32, // 浅绿
Linen	= 0xFFFFAF0E6, // 亚麻色
Magenta	= 0xFFFF00FF, // 品红
Maroon	= 0xFF800000, // 栗色
MediumAquamarine	= 0xFF66CDAA, // 中碧绿
MediumBlue	= 0xFF0000CD, // 中蓝
MediumOrchid	= 0xFFBA55D3, // 中淡紫
MediumPurple	= 0xFF9370DB, // 中紫
MediumSeaGreen	= 0xFF3CB371, // 中海绿
MediumSlateBlue	= 0xFF7B68EE, // 中岩蓝
MediumSpringGreen	= 0xFF00FA9A, // 中春绿
MediumTurquoise	= 0xFF48D1CC, // 中青绿
MediumVioletRed	= 0xFFC71585, // 中紫红
MidnightBlue	= 0xFF191970, // 午夜蓝
MintCream	= 0xFFFF5FFA, // 薄荷乳
MistyRose	= 0xFFFFE4E1, // 雾玫瑰红
Moccasin	= 0xFFFFE4B5, // 鹿皮色
NavajoWhite	= 0xFFFFDEAD, // 纳瓦霍白
Navy	= 0xFF000080, // 海军蓝
OldLace	= 0xFFFFDF5E6, // 旧鞋带
Olive	= 0xFF808000, // 橄榄
OliveDra	= 0xFF6B8E23, // 干橄榄
Orange	= 0xFFFFA500, // 橙
OrangeRed	= 0xFFFF4500, // 橙红
Orchid	= 0xFFDA70D6, // 兰花紫
PaleGoldenrod	= 0xFFEEE8AA, // 白黄
PaleGreen	= 0xFF98FB98, // 白绿
PaleTurquoise	= 0xFFAFEEEE, // 白青绿
PaleVioletRed	= 0xFFDB7093, // 白紫红
PapayaWhip	= 0xFFFFEF05, // 番木瓜鞭色
PeachPuff	= 0xFFFFFDAB9, // 桃粉

```

Peru          = 0xFFCD853F,    // 秘鲁色
Pink         = 0xFFFFC0CB,    // 粉红
Plum          = 0xFFDDA0DD,    // 梅红
PowderBlue   = 0xFFB0E0E6,    // 粉蓝
Purple        = 0xFF800080,    // 紫
Red           = 0xFFFF0000,    // 红
RosyBrown     = 0xFFBC8F8F,    // 玫瑰褐
RoyalBlue    = 0xFF4169E1,    // 皇家蓝
SaddleBrown   = 0xFF8B4513,    // 鞍褐
Salmon        = 0xFFFFA8072,   // 鲜肉色
SandyBrown   = 0xFFF4A460,    // 沙褐
SeaGreen      = 0xFF2E8B57,    // 海绿
SeaShell      = 0xFFFFF5EE,    // 海贝色
Sienna        = 0xFFA0522D,    // 赭色
Silver         = 0xFFC0C0C0,    // 银白
SkyBlue       = 0xFF87CEEB,    // 天蓝
SlateBlue     = 0xFF6A5ACD,    // 岩蓝
SlateGray     = 0xFF708090,    // 岩灰
Snow          = 0xFFFFFAFA,    // 雪白
SpringGreen   = 0xFF00FF7F,    // 春绿
SteelBlue     = 0xFF4682B4,    // 钢蓝
Tan            = 0xFFD2B48C,    // 茶色
Teal           = 0xFF008080,    // 水鸭青
Thistle        = 0xFFD8BF08,    // 菊色
Tomato         = 0xFFFF6347,    // 番茄红
Transparent   = 0x00FFFFFF,    // 透明
Turquoise     = 0xFF40E0D0,    // 宝石绿
Violet         = 0xFFEE82EE,    // 紫罗兰
Wheat          = 0xFFF5DEB3,    // 小麦色
White          = 0xFFFFFFFF,   // 白
WhiteSmoke    = 0xFFFF5F5F,    // 烟白
Yellow         = 0xFFFFFFF0,    // 黄
YellowGreen   = 0xFF9ACD32,    // 黄绿
};

// Shift count and bit mask for A, R, G, B components
enum { // 分量偏移
    AlphaShift  = 24,
    RedShift    = 16,
    GreenShift  = 8,
    BlueShift   = 0
};

enum { // 分量掩模
    AlphaMask   = 0xff000000,
    RedMask    = 0x00ff0000,

```

```
    GreenMask    = 0x0000ff00,
    BlueMask     = 0x000000ff
};

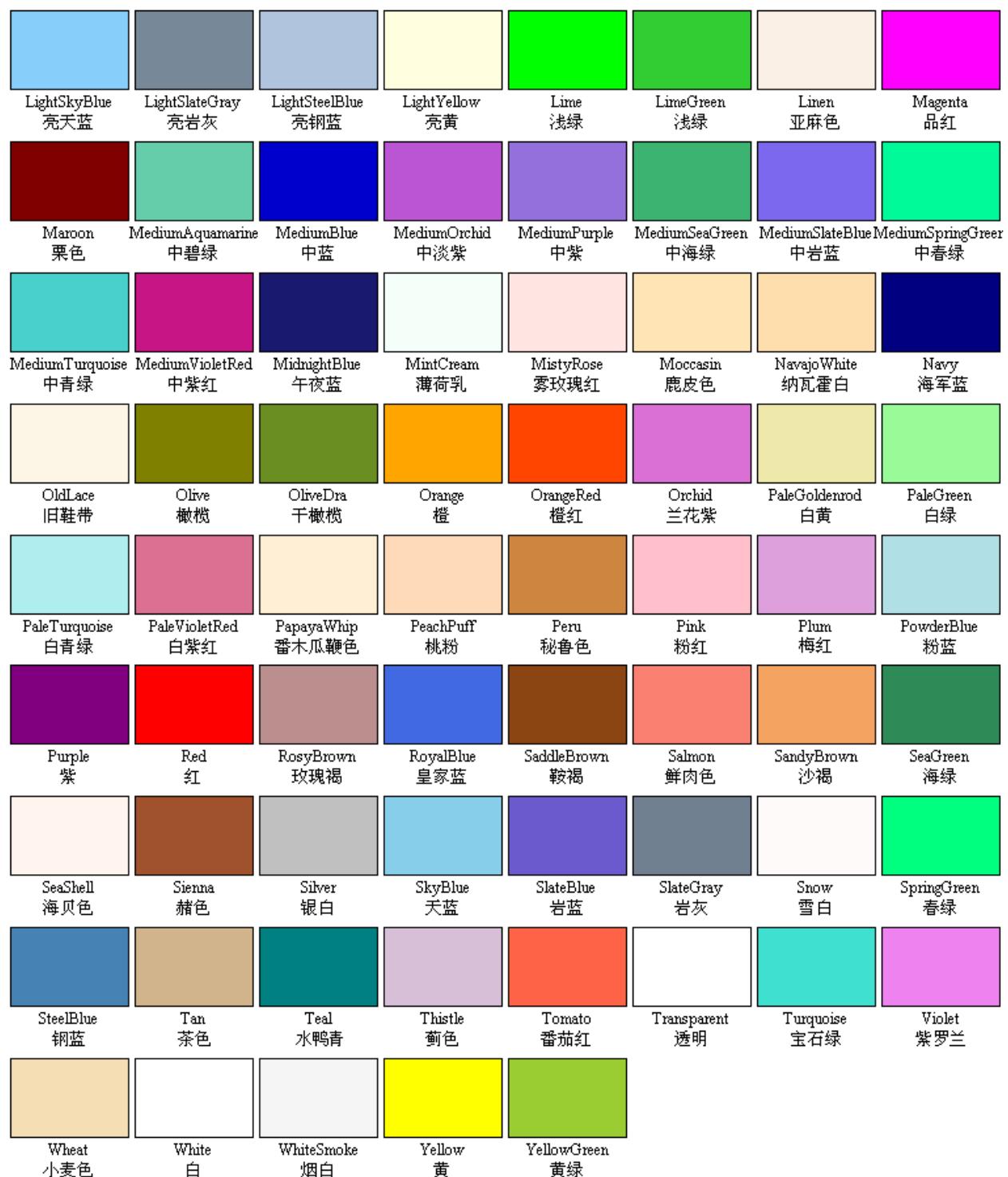
// Assemble A, R, G, B values into a 32-bit integer 装配颜色
static ARGB MakeARGB(BYTE a, BYTE r, BYTE g, BYTE b) {
    return (((ARGB)(b) << BlueShift) |
            ((ARGB)(g) << GreenShift) |
            ((ARGB)(r) << RedShift) |
            ((ARGB)(a) << AlphaShift));
}

protected:
    ARGB Argb; // 保护数据成员——颜色值 (32 位无符号整数)
};

#endif
```

下面是各种颜色枚举字符常量（包括中文译名）所对应的色块表（按字母顺序排列）：

AliceBlue 艾丽丝蓝	AntiqueWhite 古董白	Aqua 水绿	Aquamarine 碧绿	Azure 天蓝	Beige 米色	Bisque 浅黄	Black 黑
BlanchedAlmond 布兰奇杏黄	Blue 蓝	BlueViolet 蓝紫	Brown 棕褐	BurlyWood 实木	CadetBlue 军校蓝	Chartreuse 查特酒绿	Chocolate 巧克力色
Coral 珊瑚色	CornflowerBlue 矢车菊蓝	Cornsilk 玉米黄	Crimson 深红	Cyan 青	DarkBlue 暗蓝	DarkCyan 暗青	DarkGoldenrod 暗一枝黄花
DarkGray 暗灰	DarkGreen 暗绿	DarkKhaki 暗黄褐	DarkMagenta 暗品红	DarkOliveGreen 暗橄榄绿	DarkOrange 暗橙色	DarkOrchid 暗淡紫	DarkRed 暗红
DarkSalmon 暗鲜肉色	DarkSeaGreen 暗海绿	DarkSlateBlue 暗岩蓝	DarkSlateGray 暗岩灰	DarkTurquoise 暗青绿	DarkViolet 暗紫	DeepPink 深粉红	DeepSkyBlue 深天蓝
DimGray 淡灰	DodgerBlue 传单蓝	Firebrick 耐火砖色	FloralWhite 花卉白	ForestGreen 森林绿	Fuchsia 灯笼海棠紫	Gainsboro 浅灰色	GhostWhite 幽灵白
Gold 金黄	Goldenrod 一枝黄花	Gray 灰	Green 绿	GreenYellow 黄绿	Honeydew 蜜色	HotPink 热粉红	IndianRed 印度红
Indigo 靛蓝	Ivory 象牙色	Khaki 卡其黄	Lavender 淡紫（薰衣草）	LavenderBlush 紫红	LawnGreen 草坪绿	LemonChiffon 柠檬薄绸	LightBlue 亮蓝
LightCoral 亮珊瑚色	LightCyan 亮青	LightGoldenrodYellow 亮一枝黄	LightGray 亮灰	LightGreen 亮绿	LightPink 亮粉红	LightSalmon 亮鲜肉色	LightSeaGreen 亮海绿



颜色枚举常量

(你可以试着写一个输出该颜色块图的 GDI+程序。)

2) Graphics (图形)

图形类 Graphics 是 GDI+的核心，它提供绘制图形、图像和文本的各种方法（操作/函数）（似 GDI 中的 CDC 类），还可以存储显示设备和被画项目的属性（到图元文件）。Graphics

类及其成员函数都被定义在头文件 Gdiplusgraphics.h 中。

(1) 构造函数 Graphics

Graphics 类的构造函数有如下 4 种：

```
Graphics(Image* image); // 用于绘制图像  
Graphics(HDC hdc); // 用于在当前窗口中绘图  
Graphics(HDC hdc, HANDLE hdevice); // 用于在指定设备上绘制图形  
Graphics(HWND hwnd, BOOL icm = FALSE); // 用于在指定窗口中绘图  
// 可以进行颜色调整
```

其中，最常用的是第二种——在当前视图窗口中绘图的图形类构造函数。

注意，该构造函数的输入参数，是设备上下文的句柄，而不是 CDC 类对象的指针。一般可以由 CDC 对象得到 (CDC 类含有公用数据成员 HDC m_hDC;)：

- 在 OnDraw 函数中，利用输入参数 CDC *pDC，就可直接得到 DC 句柄。例如：

```
Graphics graph(pDC->m_hDC);
```
- 在视图类的其他函数中，可先利用 GetDC 函数得到 CDC 指针，然后再利用它去获取 DC 的句柄。例如：

```
Graphics graph(GetDC()->m_hDC);
```

(2) 状态枚举 status

在图形类 Graphics 中，封装了各种绘图函数。每种绘图函数被调用后，都会返回一种叫做 status 的枚举值，反映该函数是否被正确执行，0 表示正确，其他大于 0 的值为错误代码：(GdiplusTypes.h)

```
typedef enum { // 状态枚举  
    Ok = 0,  
    GenericError = 1,  
    InvalidParameter = 2,  
    OutOfMemory = 3,  
    ObjectBusy = 4,  
    InsufficientBuffer = 5,  
    NotImplemented = 6,  
    Win32Error = 7,  
    WrongState = 8,  
    Aborted = 9,  
    FileNotFound = 10,  
    ValueOverflow = 11,  
    AccessDenied = 12,  
    UnknownImageFormat = 13,  
    FontFamilyNotFound = 14,  
    FontStyleNotFound = 15,  
    NotTrueTypeFont = 16,  
    UnsupportedGdiplusVersion = 17,  
    GdiplusNotInitialized = 18,
```

```

    PropertyNotFound = 19,
    PropertyNotSupported = 20,
    ProfileNotFound = 21
} Status;

```

图形类 Graphics 中，常用的绘图函数有（先讲线型图、再讲填充图、最后讲文字）：

(3) 画直线[折线] DrawLine[s]

在 GDI+中定义了 6 种绘制直线和折线的函数：（前三个为整数版，后三个为对应的浮点数版）

```

Status DrawLine(const Pen* pen, INT x1, INT y1, INT x2, INT y2);
Status DrawLine(const Pen* pen, const Point& pt1, const Point& pt2);
Status DrawLines(const Pen* pen, const Point* points, INT count);
Status DrawLine(const Pen* pen, REAL x1, REAL y1, REAL x2, REAL y2);
Status DrawLine(const Pen* pen, const PointF& pt1, const PointF& pt2);
Status DrawLines(const Pen* pen, const PointF* points, INT count);

```

其中：

- DrawLine——画直线（4 个重载）
 - ◆ pen 为画直线所用的笔
 - ◆ (x1, y1) 和 pt1 为直线的起点
 - ◆ (x2, y2) 和 pt2 为直线的终点

GDI 的相应函数为：

```

CPoint MoveTo( int x, int y );  BOOL LineTo( int x, int y );
CPoint MoveTo( POINT point );  BOOL LineTo( POINT point );

```

- DrawLines——画折线（一串相互连接的直线段）（2 个重载）
 - ◆ points 为点数组
 - ◆ count 为数组中点的数目

GDI 的相应函数为： BOOL Polyline(LPPOINT lpPoints, int nCount);

注意，GDI+的画直线函数与 GDI 的区别：

- ◆ 笔作为绘图函数的输入参数（而不必像 GDI 中先用 SelectObject 选入 CDC）
- ◆ 一步完成绘图（而不必像 GDI 中先调用 MoveTo，再调用 LineTo）
- ◆ 有对应的浮点数版本（GDI 中无）

(4) 画矩形[组] DrawRectangle[s]

在 GDI+中也定义了 6 种绘制矩形和矩形组的函数：（也是前三个为整数版，后三个为对应的浮点数版）

```

Status DrawRectangle(const Pen* pen, const Rect& rect);
Status DrawRectangle(const Pen* pen, INT x, INT y, INT width, INT height);
Status DrawRectangles(const Pen* pen, const Rect* rects, INT count);
Status DrawRectangle(const Pen* pen, const RectF& rect);
Status DrawRectangle(const Pen* pen, REAL x, REAL y, REAL width, REAL height);

```

```
Status DrawRectangles(const Pen* pen, const RectF* rects, INT count);
```

其中：

- DrawRectangle——画单个矩形（4个重载）
 - ◆ pen 为画矩形所用的笔
 - ◆ rect 为矩形区域
 - ◆ (x, y) 为矩形的左上角
 - ◆ (width, height) 为矩形的大小（宽，高）
- DrawRectangles——画多个矩形（2个重载）
 - ◆ rects 为矩形数组
 - ◆ count 为数组中矩形的数目

注意，与 GDI 函数 Rectangle 的区别：

```
BOOL Rectangle(int x1, int y1, int x2, int y2);
```

GDI+的第2个和第4个画矩形函数的后两个输入参数，不再是 GDI 中的矩形右下角的坐标，而改成矩形的宽和高了。

另外，GDI 中没有同时绘制一个矩形数组的函数。

（5）画[椭]圆 DrawEllipse

GDI+中有 4 个重载的绘制椭圆的函数，如果输入参数所确定的外接矩形的宽高相等，则画圆。（也是前两个为整数版，后两个为对应的浮点数版）

```
Status DrawEllipse(const Pen* pen, const Rect& rect);
```

```
Status DrawEllipse(const Pen* pen, INT x, INT y, INT width, INT height)
```

```
Status DrawEllipse(const Pen* pen, const RectF& rect);
```

```
Status DrawEllipse(const Pen* pen, REAL x, REAL y, REAL width, REAL height);
```

这些函数的功能，与 GDI 中的 Ellipse 类似，但是同样要注意 GDI+与 GDI 函数的区别：

```
BOOL Ellipse( int x1, int y1, int x2, int y2 );
```

主要是，上面的第 2、4 个 GDI+画椭圆函数的后两个输入参数，也是矩形的宽高而不再是矩形的右下角坐标了。

（6）画[椭]圆弧 DrawArc

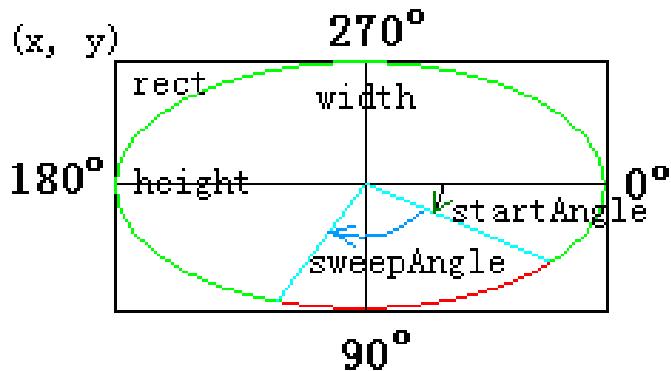
GDI+中也有 4 个重载的绘制椭圆弧的函数，如果输入参数所确定的外接矩形的宽高相等，则画圆弧。（也是前两个为整数版，后两个为对应的浮点数版）

```
Status DrawArc(const Pen* pen, INT x, INT y, INT width, INT height, REAL startAngle, REAL sweepAngle);
```

```
Status DrawArc(const Pen* pen, const Rect& rect, REAL startAngle, REAL sweepAngle);
```

```
Status DrawArc(const Pen* pen, REAL x, REAL y, REAL width, REAL height, REAL startAngle, REAL sweepAngle);
```

```
Status DrawArc(const Pen* pen, const RectF& rect, REAL startAngle, REAL sweepAngle);
```



画弧函数的输入参数

该函数的功能与 GDI 的 Arc 相同：

```
BOOL Arc( int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4 );
```

```
BOOL Arc( LPCRECT lpRect, POINT ptStart, POINT ptEnd );
```

但是也有区别，主要是，最后的参数不再是弧的终角，而是弧段所对应的扫描角。这倒是与另一个 GDI 画圆弧函数类似（其中(x, y)为圆心、nRadius 为半径、fStartAngle 为起始角、fSweepAngle 也为弧段跨角）：

```
BOOL AngleArc(int x, int y, int nRadius, float fStartAngle, float fSweepAngle);
```

当然，GDI+确定矩形的后两个参数也不再是右下角坐标，而改成宽高了，这已经是老问题了。

另外要注意的是，角度的单位是度（不是弧度，C++的三角函数采用的是弧度单位），而且都必须是实数。零度角为 x 轴方向，顺时针方向为正（这与数学上反时针方向为正刚好相反）。

(7) 画多边形 DrawPolygon

GDI+中有 2 个重载的绘制多边形的函数，第一个为整数版，后一个为对应的浮点数版：

```
Status DrawPolygon(const Pen* pen, const Point* points, INT count);
```

```
Status DrawPolygon(const Pen* pen, const PointF* points, INT count);
```

其中，各参数的含义同画折线函数 DrawLines 的，只是 DrawPolygon 函数会将点数组中的起点和终点连接起来，形成一个封闭的多边形区域。

该函数的功能与 GDI 的 Polygon 相同：

```
BOOL Polygon( LPPOINT lpPoints, int nCount );
```

注意：GDI+中没有提供，与 GDI 函数 RoundRect（圆角矩形）和 Chord（弓弦），具有类似功能的绘图函数。可以利用矩形+椭圆和弧+直线等函数自己来实现。

(8) 画填充图

在 GDI+中画填充图，不需像 GDI 那样得先将刷子选入 DC，而是与 GDI+画线状图的函数类似，将刷子作为画填充图函数的第一个输入参数。

- 画填充矩形[组] FillRectangle[s]

```
Status FillRectangle(const Brush* brush, const Rect& rect);
```

```
Status FillRectangle(const Brush* brush, INT x, INT y, INT width, INT height);
```

```
Status FillRectangles(const Brush* brush, const Rect* rects, INT count);
Status FillRectangle(const Brush* brush, const RectF& rect);
Status FillRectangle(const Brush* brush, REAL x, REAL y, REAL width, REAL
height);
```

```
Status FillRectangles(const Brush* brush, const RectF* rects, INT count);
```

用指定刷子 Brush, 填充 rect 的内部区域, 无边线, 填充区域包括矩形的左边界和上边界, 但不包括矩形的右边界和下边界。功能与 GDI 的 FillRect 类似:

```
void FillRect( LPCRECT lpRect, CBrush* pBrush );
```

但是, GDI 中没有同时填充一个矩形数组的函数。不过 GDI 却有 GDI+没有的画填充圆角矩形的函数 FillSolidRect。

- 画填充[椭]圆 **FillEllipse**

```
Status FillEllipse(const Brush* brush, const Rect& rect);
```

```
Status FillEllipse(const Brush* brush, INT x, INT y, INT width, INT height);
```

```
Status FillEllipse(const Brush* brush, const RectF& rect);
```

```
Status FillEllipse(const Brush* brush, REAL x, REAL y, REAL width, REAL height);
```

GDI 中没有类似函数, 但可以用 (采用当前刷填充的) Ellipse 来代替。

- 画饼图 **DrawPie**

```
Status DrawPie(const Pen* pen, const Rect& rect, REAL startAngle, REAL
sweepAngle);
```

```
Status DrawPie(const Pen* pen, INT x, INT y, INT width, INT height, REAL startAngle,
REAL sweepAngle);
```

```
Status DrawPie(const Pen* pen, const RectF& rect, REAL startAngle, REAL
sweepAngle);
```

```
Status DrawPie(const Pen* pen, REAL x, REAL y, REAL width, REAL height, REAL
startAngle, REAL sweepAngle);
```

与 GDI 的下列函数类似, 但是部分输入参数的含义有所不同:

```
BOOL Pie( int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4 );
```

```
BOOL Pie( LPCRECT lpRect, POINT ptStart, POINT ptEnd );
```

例如:

```
void DrawPies(Graphics &graph, const Color cols[], Point &O, int r, const float data[],
int n) {
```

```
    Rect rect(O.X - r, O.Y - r, 2 * r, 2 * r);
```

```
    float startAngle = 0, sweepAngle;
```

```
    for (int i = 0; i < n; i++) {
```

```
        sweepAngle = data[i] * 360.0f;
```

```
        graph.FillPie(new SolidBrush(cols[i]), rect, startAngle, sweepAngle);
```

```
        startAngle += sweepAngle;
```

```
}
```

```
}
```

```
void CGdipDrawView::OnDraw(CDC* pDC) {
```

```
    ....
```

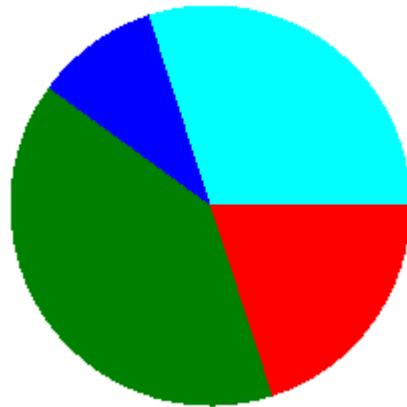
```
    Graphics graph(pDC->m_hDC);
```

```
    Color cols[] = {Color::Red, Color::Green, Color::Blue, Color::Aqua};
```

```
    float data[] = {0.2f, 0.4f, 0.1f, 0.3f};
```

```
DrawPies(graph, cols, Point(200, 200), 100, data, 4);
.....
}
```

输出结果为：



饼图

- 画填充多边形 **FillPolygon**

```
Status FillPolygon(const Brush* brush, const Point* points, INT count);
Status FillPolygon(const Brush* brush, const Point* points, INT count, FillMode
fillMode);
Status FillPolygon(const Brush* brush, const PointF* points, INT count);
Status FillPolygon(const Brush* brush, const PointF* points, INT count, FillMode
fillMode);
```

GDI 中也没有类似函数，但可以用（采用当前刷填充的）**Polygon** 来代替。

其中，填充模式参数 **FillMode**，可取如下两个值之一：

```
typedef enum {
    FillModeAlternate, // 交替模式——按奇偶规则填充
    FillModeWinding // 环绕模式——按非零环绕规则填充
} FillMode;
```

对简单图形，这两种模式的效果是一样的，但对复杂图形，特别是有穿插的图，结果可能是不同的。例如：（画五角星）

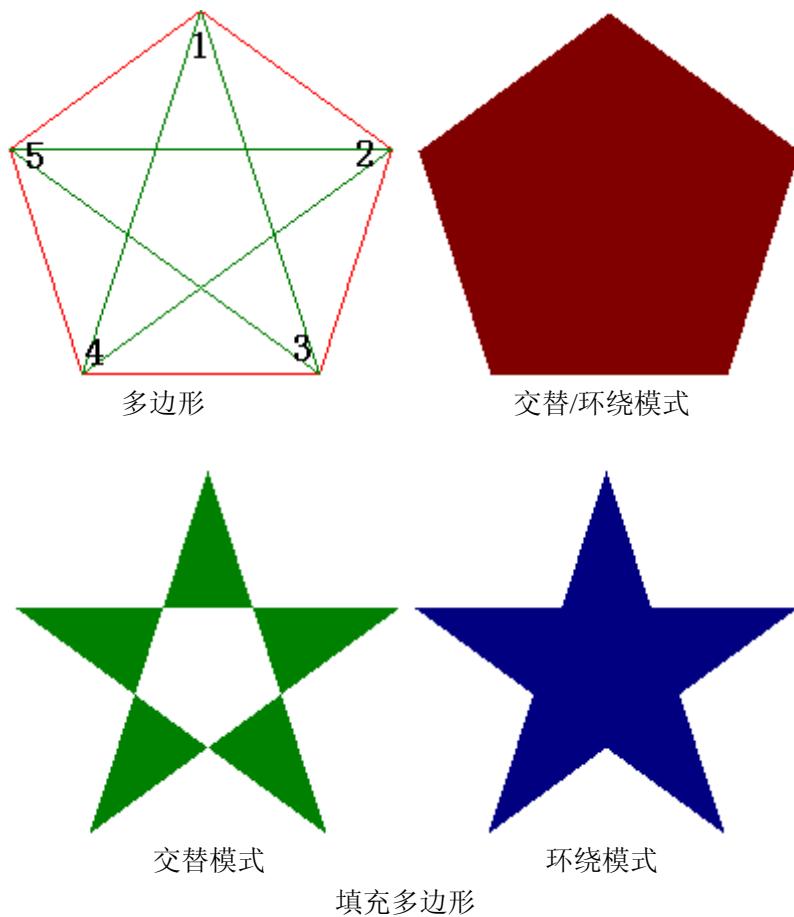
```
void Example_FillPolygon(HDC hdc) {
    const int n = 5;
    Point point1(100, 0);
    Point point2(195, 69);
    Point point3(159, 181);
    Point point4(41, 181);
    Point point5(5, 69);
    Point points0[n] = {point1, point2, point3, point4, point5};
    Point points[n] = {point1, point3, point5, point2, point4};
    // Create SolidBrush object.
    //SolidBrush brush(Color(128, 0, 0));
    SolidBrush brush(Color(0, 128, 0));
    //SolidBrush brush(Color(0, 0, 128));
    // Fill the polygon.
```

```

Graphics graphics(hdc);
//graphics.DrawPolygon(new Pen(Color::Red), points0, n);
//graphics.DrawPolygon(new Pen(Color::Green), points, n);
//graphics.FillPolygon(&brush, points0, n, FillModeAlternate);
graphics.FillPolygon(&brush, points, n, FillModeAlternate);
//graphics.FillPolygon(&brush, points, n, FillModeWinding);
}

```

输出结果如：



注意，在 GDI 中，任何画封闭区域的性状图绘制函数（如矩形、圆角矩形、[椭]圆、弓弦和多边形等），都可以画填充图，因为它们总是在用当前笔画指定边框的同时，也用当前刷子填充内部区域。

而 GDI+的画线函数就没有这个功能，因为在 GDI+是无状态的，没有当前笔和刷的概念。为了完成与这些 GDI 函数类似的功能，在 GDI+中，你得分两步来做：先用填充函数填充区域内部，再用画线函数绘制边框。

(9) 画曲线

前面讲的各种画线状图或填充图的 GDI+函数，虽然在形式上与 GDI 的有所不同（函数名前加了 Draw 或 Fill、将笔或刷作为第一个输入参数、部分位置输入参数改成了大小参数，并增加了浮点数版），但是在功能上却是相同的。

现在要讲的曲线绘制，则是 GDI+新增加的内容。曲线在机械设计、工程建筑和图形动

画等领域，都有十分广泛应用。

常用的曲线有 Bezier（贝塞尔）曲线和样条（spline）曲线。贝塞尔曲线比较简单，适合于画控制点少的曲线。当控制点太多时，要不曲线的次数（比点数少 1）太高，要不拼接比较困难，而且没有局部性（即修改一点影响全局），性能不太好。而样条曲线则可以画任意多个控制点的曲线，曲线的次数也可以指定（一般为二次或三次，如 TrueType 字体采用的是二次 B 样条曲线），并且具有局部性。贝塞尔曲线特别是样条曲线有很多变种。常见的贝塞尔曲线有普通贝塞尔曲线和有理贝塞尔曲线。常用的样条曲线有：B 样条、 β 样条、Hermite（厄密）样条、基数样条、Kochanek-Bartels 样条和 Catmull-Rom 样条等。

GDI+中所实现的是普通贝塞尔曲线（不过控制点，位于控制多边形的凸包之内）和基数样条曲线（过控制点）。

有关曲线和曲面构造方法，会在课程《计算机图形学》中介绍。

● 基数样条曲线 (cardinal spline curve)

```
Status DrawCurve(const Pen* pen, const Point* points, INT count); // tension = 0.5f
Status DrawCurve(const Pen* pen, const Point* points, INT count, REAL tension);
Status DrawCurve(const Pen* pen, const Point* points, INT count, INT offset, INT
    numberofSegments, REAL tension = 0.5f); // 只画部分点
Status DrawCurve(const Pen* pen, const PointF* points, INT count);
Status DrawCurve(const Pen* pen, const PointF* points, INT count, REAL tension);
Status DrawCurve(const Pen* pen, const PointF* points, INT count, INT offset, INT
    numberofSegments, REAL tension = 0.5f);
Status DrawClosedCurve(const Pen* pen, const Point* points, INT count);
Status DrawClosedCurve(const Pen *pen, const Point* points, INT count, REAL
    tension);
Status DrawClosedCurve(const Pen* pen, const PointF* points, INT count);
Status DrawClosedCurve(const Pen *pen, const PointF* points, INT count, REAL
    tension);
```

其中：

参数 tension（张力）指定曲线的弯曲程度，tension = 0.0（直线）~ 1.0（最弯曲）

无张力版的函数的 tension = 0.5（缺省值）

第 3/6 个 DrawCurve，只画从 points[offset]开始的 numberofSegments 个点组成的部分曲线段

DrawClosedCurve 函数（连接首尾点）画封闭的基数样条曲线

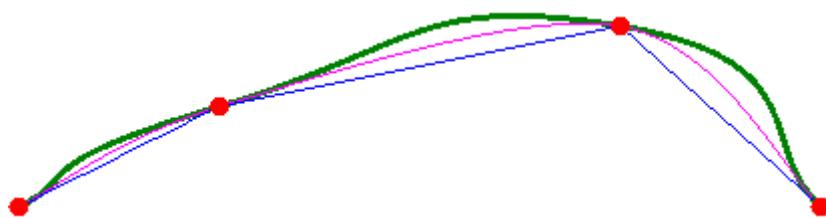
例如：

```
void DrawPoints(Graphics &graph, const Color &col, int r, const Point* points, INT
    count) {
    SolidBrush brush(col);
    for (int i = 0; i < count; i++)
        graph.FillEllipse(&brush, Rect(points[i].X - r, points[i].Y - r, 2 * r, 2 * r));
}
void Example_DrawCurve(HDC hdc) {
    Graphics graphics(hdc);
    // Define a Pen object and an array of Point objects.
    Pen greenPen(Color::Green, 3);
    Point point1(100, 100);
    Point point2(200, 50);
```

```

Point point3(400, 10);
Point point4(500, 100);
Point curvePoints[4] = {point1, point2, point3, point4};
// Draw the curve.
graphics.DrawCurve(&greenPen, curvePoints, 4, 1.0);
graphics.DrawCurve(new Pen(Color::Magenta), curvePoints, 4, 0.5);
graphics.DrawCurve(new Pen(Color::Blue), curvePoints, 4, 0.0);
//graphics.DrawCurve(&greenPen, curvePoints, 4);
//graphics.DrawClosedCurve(new Pen(Color::Aqua), curvePoints, 4);
//graphics.DrawBeziers(new Pen(Color::Chocolate), curvePoints, 4);
DrawPoints(graphics, Color::Red, 5, curvePoints, 4); // Draw the points in the curve.
}

```



基数样条曲线

tension = 0.0 (蓝色细折线的)、0.5 (红色细平坦曲线)、1.0 (绿色粗弯曲曲线)



曲线

基数样条(绿色)、封闭基数样条(水绿色)、贝塞尔(巧克力色)

● 贝塞尔曲线 (Bezier curve)

```

Status DrawBezier(const Pen* pen, INT x1, INT y1, INT x2, INT y2, INT x3, INT y3,
                  INT x4, INT y4);
Status DrawBezier(const Pen* pen, const Point& pt1, const Point& pt2, const Point&
                  pt3, const Point& pt4);
Status DrawBeziers(const Pen* pen, const Point* points, INT count);
Status DrawBezier(const Pen* pen, REAL x1, REAL y1, REAL x2, REAL y2, REAL x3,
                  REAL y3, REAL x4, REAL y4);
Status DrawBezier(const Pen* pen, const PointF& pt1, const PointF& pt2, const
                  PointF& pt3, const PointF& pt4);
Status DrawBeziers(const Pen* pen, const PointF* points, INT count);

```

● 填充封闭基数样条曲线

```

Status FillClosedCurve(const Brush* brush, const Point* points, INT count);
Status FillClosedCurve(const Brush* brush, const Point* points, INT count, FillMode

```

```

fillMode, REAL tension = 0.5f);
Status FillClosedCurve(const Brush* brush, const PointF* points, INT count);
Status FillClosedCurve(const Brush* brush, const PointF* points, INT count, FillMode
fillMode, REAL tension = 0.5f);

```

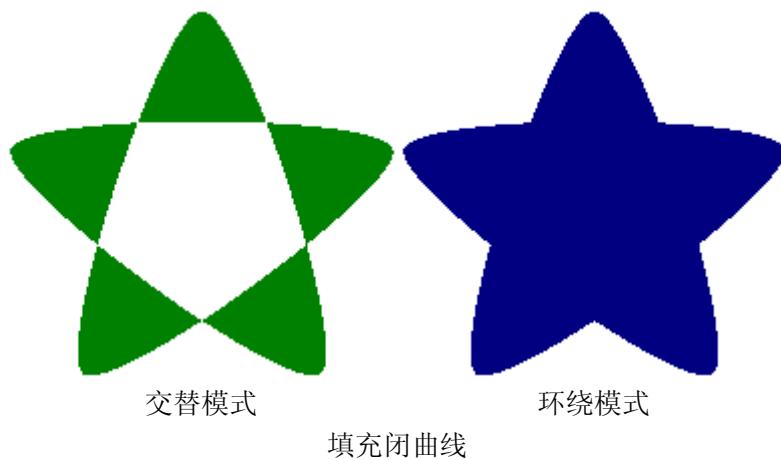
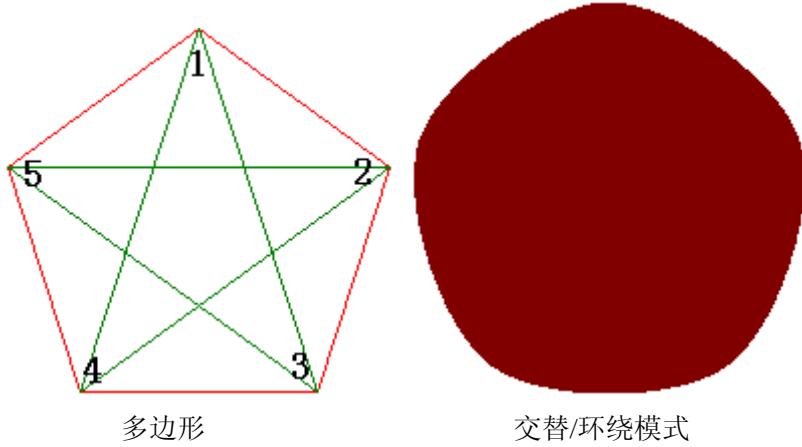
例如：(将前面的多边形填充例子中的填充函数，改为填充封闭曲线函数即可)

```

graphics.FillClosedCurve(&brush, points, n, FillModeAlternate);
graphics.FillClosedCurve(&brush, points, n, FillModeWinding);

```

输出结果如：



(10) 平滑

可以利用 Graphics 类的设置平滑模式成员函数

```

Status SetSmoothingMode(SmoothingMode smoothingMode);

```

来设置绘图时的平滑化处理。其中的输入参数为枚举类型

```

typedef enum {
    SmoothingModeInvalid = QualityModeInvalid, // 无效 (保留)
    SmoothingModeDefault = QualityModeDefault, // 缺省 (低质, 无平滑处理)
    SmoothingModeHighSpeed = QualityModeLow, // 高速 (低质, 无平滑处理)
    SmoothingModeHighQuality = QualityModeHigh, // 高质 (使用 8*4 盒过滤器)
    SmoothingModeNone, // 无平滑处理
    SmoothingModeAntiAlias8x4, // 使用 8*4 盒过滤器 (库中无)
}

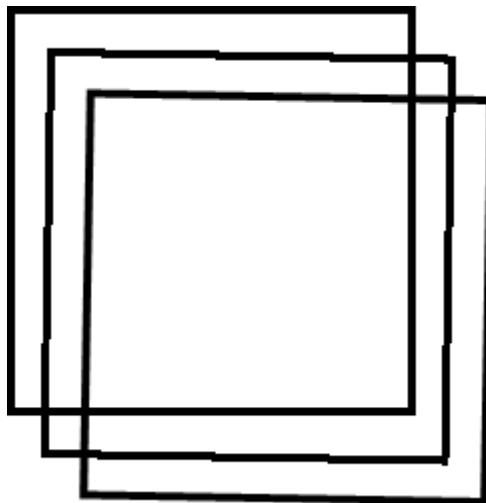
```

```
SmoothingModeAntiAlias = SmoothingModeAntiAlias8x4, // 使用 8*4 盒过滤器
SmoothingModeAntiAlias8x8 // 使用 8*8 盒过滤器（最高质，库中也无）
} SmoothingMode;
```

例如：

```
Graphics graph(pDC->m_hDC);
Pen pen(Color::Black, 4);
Rect rect(10, 10, 200, 200);
graph.DrawRectangle(&pen, rect);
graph.RotateTransform(1);
graph.TranslateTransform(20, 20);
//graph.SetSmoothingMode(SmoothingModeNone);
graph.DrawRectangle(&pen, rect);
graph.TranslateTransform(20, 20);
graph.SetSmoothingMode(SmoothingModeAntiAlias);
graph.DrawRectangle(&pen, rect);
```

输出结果为：



(11) 清屏 Clear

GDI 中没有用于清屏的专门函数，得自己用背景色画窗口大小的填充矩形，或者调用窗口类的 Invalidate 和 UpdateWindow 函数。现在，GDI+有了清屏函数 Clear：

```
Status Clear(const Color &color);
```

其中的输入参数 color，为用户指定的填充背景色。例如：

```
Graphics graph(GetDC()->m_hDC);
graph.Clear(Color::White);
```

3) Pen (笔)

与 GDI 中的一样，GDI+中的笔（pen 钢笔/画笔）也是画线状图的工具，但是功能更加

强大。例如：透明笔、图案笔、自定义虚线风格、线帽、笔的缩放和旋转、笔的连接点属性等。

GDI+中的笔对应于 Pen 类，被定义在 GdiplusPen.h 头文件中。

笔的构造函数主要有两个：

```
Pen(const Color &color, REAL width = 1.0); // 单色笔  
Pen(const Brush *brush, REAL width = 1.0); // 纹理图案笔
```

其中，最常用的是第一个，它构造一个颜色为 color，宽度为 width（缺省为 1）的单色笔。如果颜色的 α 值<255，则所创建的笔就是带透明度的笔。

(0) 笔对齐

当笔宽大于 1 时，缺省情况下，是以笔的中心与绘图坐标对齐。但是，也可以采用 Pen 类的成员函数：

```
Status SetAlignment(PenAlignment penAlignment);
```

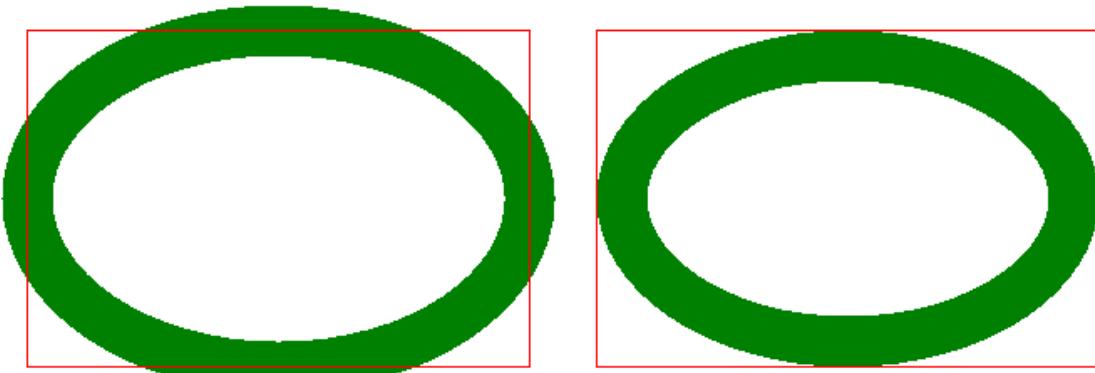
设置为内对齐，其输入参数取枚举类型 PenAlignment 的符号常量：

```
typedef enum {  
    PenAlignmentCenter = 0, // 中心对齐（缺省值）  
    PenAlignmentInset = 1 // 内对齐  
} PenAlignment;
```

例如：

```
Graphics graph(pDC->m_hDC);  
Rect rect(20, 20, 300, 200);  
Pen pen(Color::Green, 30), redPen(Color::Red);  
graph.DrawEllipse(&pen, rect);  
graph.DrawRectangle(&redPen, rect);  
pen.SetAlignment(PenAlignmentInset);  
graph.TranslateTransform(340, 0);  
graph.DrawEllipse(&pen, rect);  
graph.DrawRectangle(&redPen, rect);
```

输出结果为：



中心对齐（缺省值）

笔对齐

内对齐

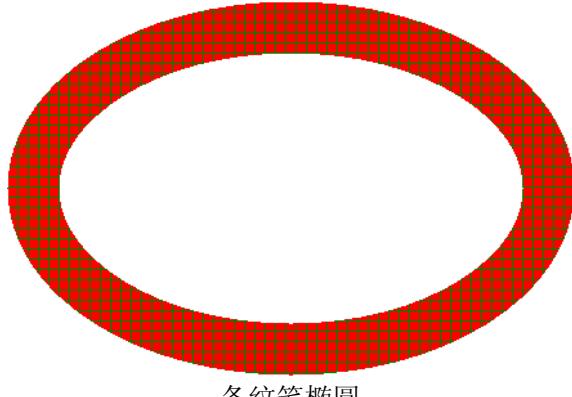
(1) 图案笔

笔类 Pen 的第二个构造函数，是从刷子来创建笔，如果是单色的实心刷，则相当于第一个笔构造函数。如果刷子为影线（条纹）或纹理（图像）等图案刷，则该构造函数所创见的就是对应的图案笔。

例如：(条纹笔画椭圆)

```
HatchBrush hBrush(HatchStyleCross, Color::Green, Color::Red); // 创建十字线影线  
刷  
Pen hPen(&hBrush, 40); // 创建宽度为 40 像素的影线笔  
graph.DrawEllipse(&hPen, 20, 20, 400, 250); // 画椭圆
```

输出结果如：

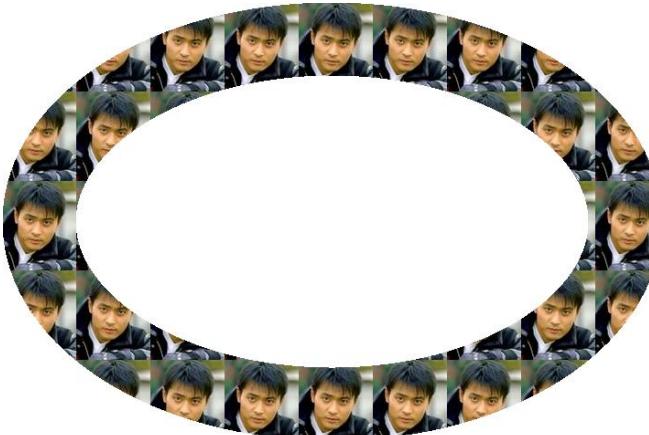


条纹笔椭圆

又例如：(纹理笔画椭圆)

```
Image img(L"张东健.bmp"); // 创建图像对象，并装入图像文件  
TextureBrush tBrush(&img); // 创建纹理刷  
Pen tPen(&tBrush, 80); // 创建宽度为 80 像素的纹理笔  
Graphics graph(GetDC()->m_hDC); // 创建图形对象  
graph.DrawEllipse(&tPen, 40, 40, 640, 400); // 画椭圆
```

输出结果如：



纹理笔椭圆

(2) 线型

与 GDI 一样，对 GDI+中的笔，也可以设置线型。所用的成员函数为：

```
Status SetDashStyle(DashStyle dashStyle);
```

其中的输入参数，为虚线风格枚举 DashStyle：(GdiplusEnums.h)

```
enum DashStyle {
    DashStyleSolid,      // 0 实线:  (缺省值)
    DashStyleDash,        // 1 虚线: 
    DashStyleDot,         // 2 点线: 
    DashStyleDashDot,     // 3 虚点线: 
    DashStyleDashDotDot, // 4 虚点点线: 
    DashStyleCustom       // 5 自定义虚线: 
};
```

可以用 Pen 类的另一个成员函数来获取笔的线型：

```
DashStyle GetDashStyle() const;
```

GDI+中的线型，大多数与 GDI 中的相同，区别主要有两点：

- GDI 中的非实线线型，对宽度>1 的笔无效；而 GDI+的笔对任意非零宽度的笔都是有效的；
- GDI+中新增了一种风格——自定义虚线风格。

具体的自定义虚线风格，由 Pen 类的设置虚线图案的成员函数

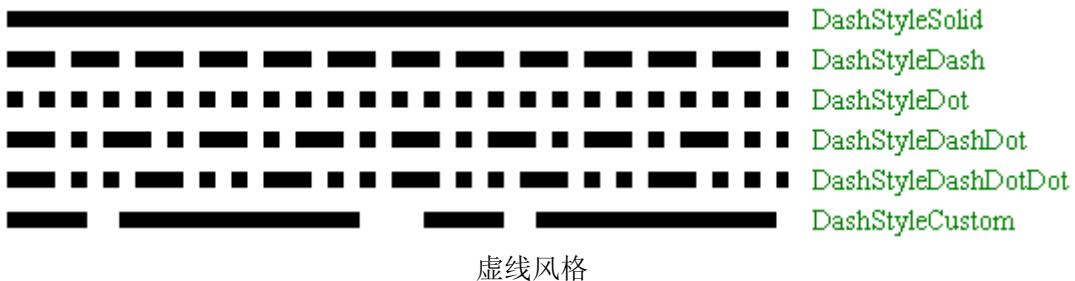
```
Status SetDashPattern(const REAL *dashArray, INT count);
```

来设置，其中的实数数组 dashArray 含若干个正实数（单位为像素），按线、空、线、空、……的交叉方式排列；参数 count 为数组中实数的个数（须>0）。

例如：

```
Graphics graph(pDC->m_hDC);
Pen pen(Color::Black, 8); // 创建宽 8 个像素的黑色笔（画虚线用）
REAL dashVals[4] = {5.0f, 2.0f, 15.0f, 4.0f}; // 线 5、空 2、线 15、空 4（像素）
FontFamily fontFamily(L"Times New Roman"); // 创建字体族对象
Font font(&fontFamily, 10.5); // 创建 5 号字大小的 Times New Roman 字体
SolidBrush brush(Color(0, 128, 0)); // 创建绿色的实心刷（写字符串用）
// 笔的虚线风格枚举常量的名称字符串数组
CString strs[] = {L"DashStyleSolid", L"DashStyleDash", L"DashStyleDot",
                  L"DashStyleDashDot", L"DashStyleDashDotDot", L"DashStyleCustom"};
for (int i = 0; i <= 5; i++) { // 绘制各种风格的虚线及其名称串
    pen.SetDashStyle((DashStyle)i); // 设置笔的虚线风格
    if (i == 5) pen.SetDashPattern(dashVals, 4); // 设置自定义虚线图案
    graph.DrawLine(&pen, 10, 10 + i * 20, 400, 10 + i * 20); // 画虚线
    // 绘制虚线风格枚举常量名称字符串
    graph.DrawString(strs[i], -1, &font, PointF(410, 2 + i * 20), &brush);
}
```

输出结果如：



虚线风格

还可以用 Pen 类的另一个成员函数来获取笔的自定义虚线图案数据：

```
INT GetDashPatternCount(VOID); // 获取虚线数组中实数的个数
Status GetDashPattern(REAL *dashArray, INT count); // 获取虚线数组
```

(3) 线帽

线帽 (line cap) 是指线条两端的外观，缺省为正方形，也可以用 Pen 类的下列成员函数来设置不同的线端形状：

```
Status SetStartCap(LineCap startCap); // 设置起点的线帽
Status SetEndCap(LineCap endCap); // 设置终点的线帽
// 设置起点、终点和虚线的线帽
Status SetLineCap(LineCap startCap, LineCap endCap, DashCap dashCap);
```

其中的线帽枚举 LineCap 为：(GdiplusEnums.h)

```
typedef enum {
    LineCapFlat = 0, // 平线，直线起点位于平线的中点（缺省值）
    LineCapSquare = 1, // 方形，高度=线宽，直线起点位于正方形中心
    LineCapRound = 2, // 圆形，直径=线宽，直线起点位于圆心
    LineCapTriangle = 3, // 三角，高度=线宽，直线起点位于其底边中点
    LineCapNoAnchor = 0x10, // 无锚，同平线
    LineCapSquareAnchor = 0x11, // 方形锚，高度>线宽，直线起点位于正方形中心
    LineCapRoundAnchor = 0x12, // 圆形锚，直径>线宽，直线起点位于圆心
    LineCapDiamondAnchor = 0x13, // 菱形锚，高度>线宽，直线起点位于菱形中心
    LineCapArrowAnchor = 0x14, // 箭头锚，高度>线宽，直线起点位于箭头的尖点
    LineCapCustom = 0xff // 自定义线帽
} LineCap;
```

自定义线帽，需要用到 GDI+专门为此定义的类 CustomLineCap。其构造函数为：

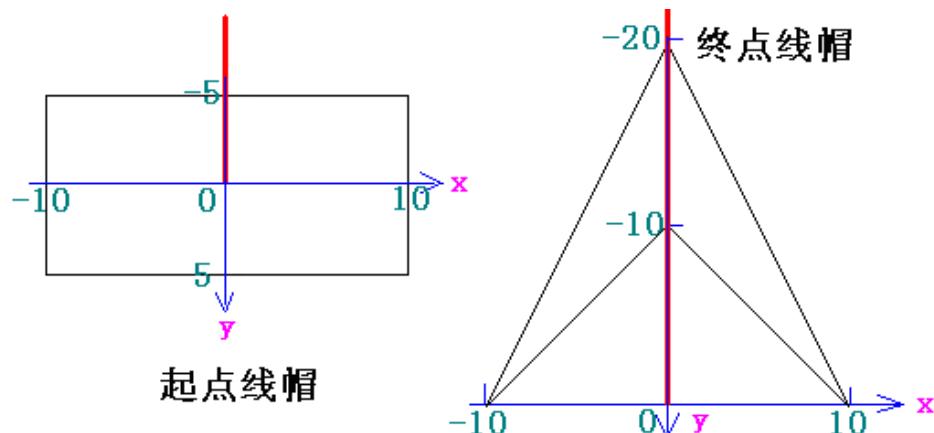
```
CustomLineCap(const GraphicsPath *fillPath, const GraphicsPath *strokePath, LineCap
baseCap, REAL baseInset);
```

其中要用到图形路径类 GraphicsPath，该类中有图形各种添加图形函数，只是把 Graphics 类绘图函数名中的 Draw 改成 Add 即可。例如：AddLine、AddRectangle 和 AddPolygon 等。使用时，可以先创建一个空路径，然后调用这些添加图形函数若干次，就可以生成路径了。

例如：



自定义线帽



构造自定义线帽头尾所使用的坐标系

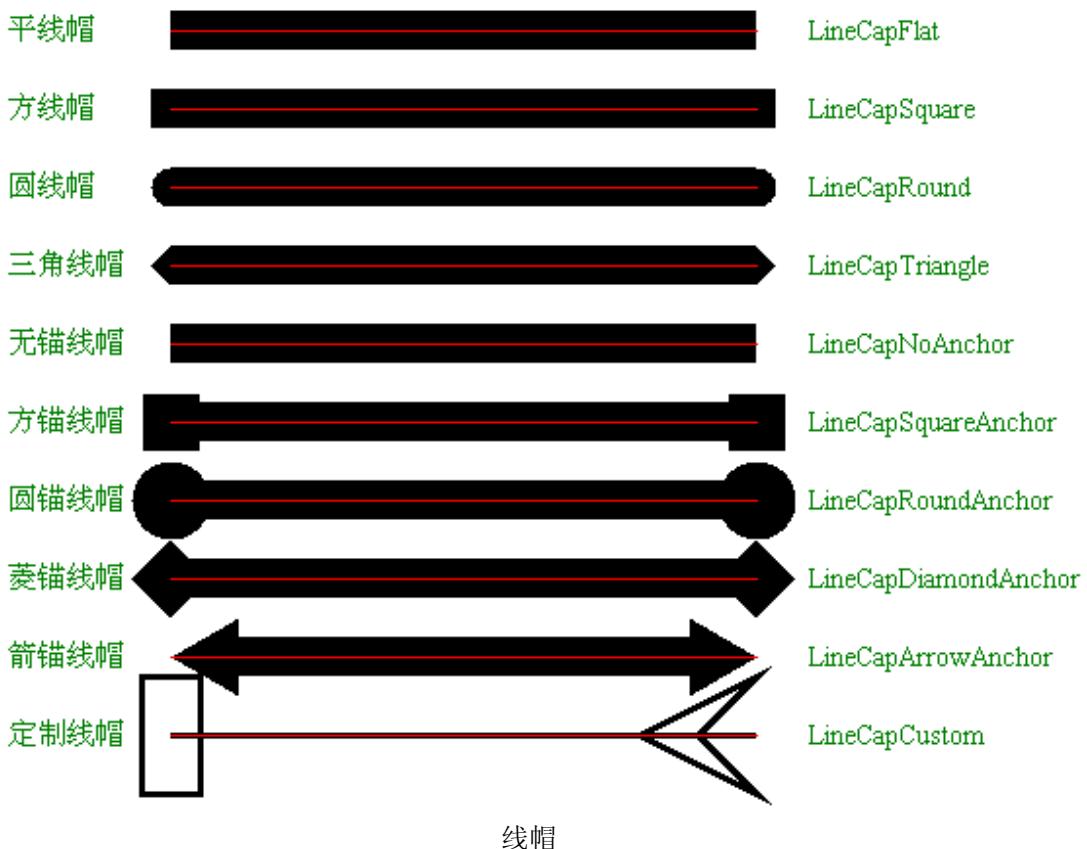
```

// 自定义线帽
GraphicsPath startPath, endPath; // 创建起点和终点路径对象
startPath.AddRectangle(Rect(-10, -5, 20, 10)); // 起点矩形
Point polygonPoints[4] = {Point(0, -20), Point(10, 0), Point(0, -10), Point(-10, 0)};
endPath.AddPolygon(polygonPoints, 4); // 终点箭头
CustomLineCap startCap(NULL, &startPath); // 创建起点线帽
CustomLineCap endCap(NULL, &endPath); // 创建终点线帽
// 定义笔
Pen pen(Color::Black, 20); // 画带线帽粗线的黑笔
Pen redPen(Color::Red); // 画不带线帽细线的红笔
// 中英文线帽字符串数组
CString cstrs[] = {L"平线帽", L"方线帽", L"圆线帽", L"三角线帽", L"无锚线帽",
    L"方锚线帽", L"圆锚线帽", L"菱锚线帽", L"箭锚线帽", L"定制线帽"};
CString estrs[] = {L"LineCapFlat", L"LineCapSquare", L"LineCapRound",
    L"LineCapTriangle", L"LineCapNoAnchor", L"LineCapSquareAnchor",
    L"LineCapRoundAnchor", L"LineCapDiamondAnchor",
    L"LineCapArrowAnchor", L"LineCapCustom"};
FontFamily fontFamily(L"Times New Roman"); // 或"宋体"
Font font(&fontFamily, 10.5); // 五号字
// 绘制各种线帽
Graphics graph(pDC->m_hDC);
for (int i = 0; i <= 9; i++) { // 画线循环
    LineCap lc = (LineCap)(i < 4 ? i : i + 12); // 线帽常量（整数）
    if (i < 9) pen.SetLineCap(lc, lc, DashCapFlat); // 标准线帽
}

```

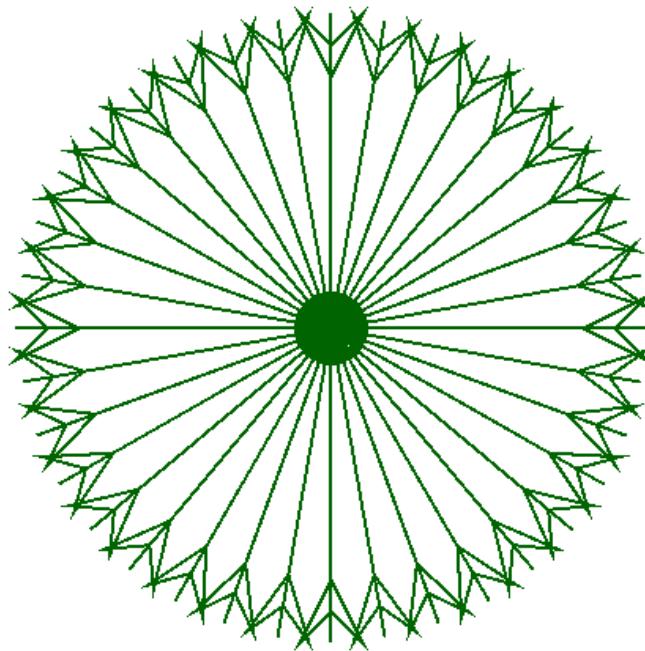
```
else { // 自定义线帽 (i = 9)
    pen.SetCustomStartCap(&startCap); // 设置自定义的起点线帽
    pen.SetCustomEndCap(&endCap); // 设置自定义的终点线帽
    pen.SetWidth(3.0f); // 重新设置线宽为 3 个像素
}
int y = 20 + i * 40; // 计算直线的垂直坐标
graph.DrawLine(&pen, 100, y, 400, y); // 画带线帽的粗线
graph.DrawLine(&redPen, 100, y, 400, y); // 画不带线帽的细线
// 绘制中英文线帽字符串
graph.DrawString(cstrs[i], -1, &font, PointF(15.0f, y - 8.0f), &brush);
graph.DrawString(estrs[i], -1, &font, PointF(425.0f, y - 8.0f), &brush);
}
```

输出结果如：



又例如：（startCap 和 endCap 的创建同上例。需包含头文件 <math.h>）

输出结果为：



自定义线帽的旋转直线簇

函数 SetLineCap 的最后一个输入参数 DashCap dashCap，用于设置虚线内部各线段端点的形状。其取值是枚举类型：(GdiplusEnums.h)

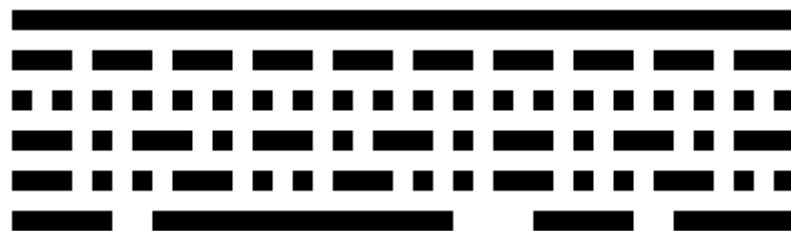
```
typedef enum {
    DashCapFlat = 0, // 平线（缺省值）
    DashCapRound = 2, // 圆形
    DashCapTriangle = 3 // 三角
} DashCap;
```

可见，只有三种选择：平、圆和三角。之所以枚举常量所对应的值不连续，是因为要同 LineCap 枚举的对应常量一致。

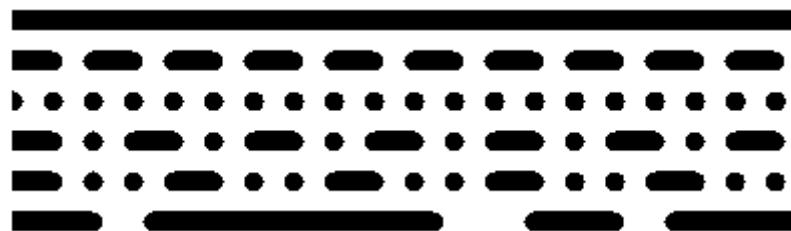
注意，虚线帽的设置，只影响其虚线内部的线段，不会影响整条虚线的头尾形状，它们是由 SetLineCap 函数的前两个参数来分别设置的。例如：

```
Graphics graph(pDC->m_hDC);
Pen pen(Color::Black, 10);
pen.SetLineCap(LineCapFlat, LineCapFlat, DashCapFlat);
//pen.SetLineCap(LineCapFlat, LineCapFlat, DashCapRound);
//pen.SetLineCap(LineCapFlat, LineCapFlat, DashCapTriangle);
REAL dashVals[4] = { 5.0f, 2.0f, 15.0f, 4.0f };
for (int i = 0; i <= 5; i++) {
    pen.SetDashStyle((DashStyle)i);
    if (i == 5) pen.SetDashPattern(dashVals, 4);
    graph.DrawLine(&pen, 10, 10 + i * 20, 400, 10 + i * 20);
}
```

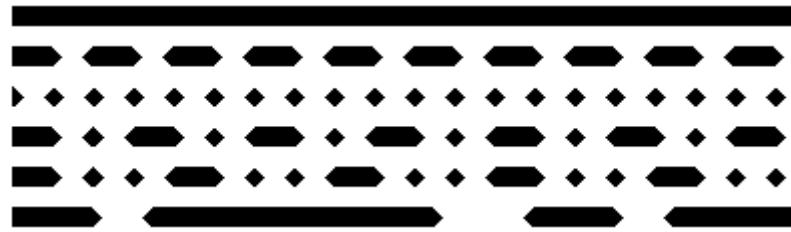
输出结果如：



DashCapFlat 平虚线帽



DashCapRound 圆虚线帽



DashCapTriangle 三角虚线帽

(4) 线连接

笔的线连接属性，也是 GDI+新增的功能。可以使用 Pen 类的成员函数 LineJoin:

```
Status SetLineJoin(LineJoin lineJoin);
```

来设置笔的线连接属性。其中输入参数为枚举类型 LineJoin:

```
enum LineJoin {  
    LineJoinMiter      = 0, // 斜接 (缺省值)  
    LineJoinBevel      = 1, // 斜截  
    LineJoinRound      = 2, // 圆角  
    LineJoinMiterClipped = 3 // 斜剪  
};
```

例如：

```
Graphics graph(pDC->m_hDC);  
Pen pen(Color::DarkGreen, 40);  
for (int i = 0; i < 4; i++) {  
    pen.SetLineJoin((LineJoin)i);  
    graph.DrawRectangle(&pen, 40 + i * 150, 40, 100, 100);  
}
```

输出结果为：



从该例还看不出斜剪与斜接有什么区别，因为 LineJoinMiterClipped 主要针对交角很小，相交部分很长的情形。在斜剪线连接方式下，可以调用 Pen 类的成员函数

```
Status SetMiterLimit(REAL miterLimit);
```

来设置相交部分的最大限制长度，缺省是 10.0（相对于线宽的比值）。

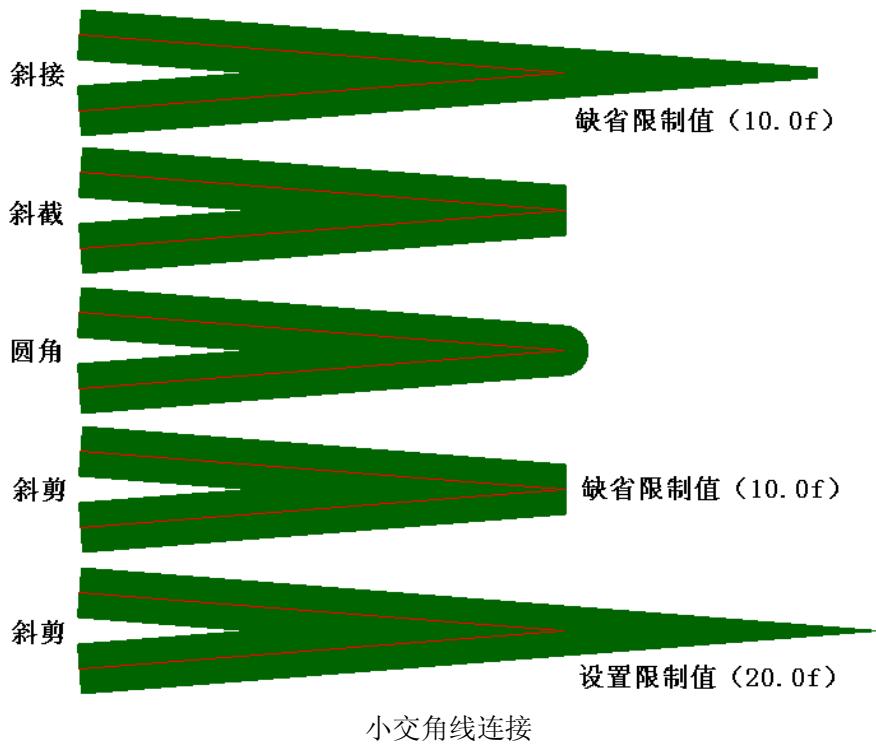
对 LineJoinMiterClipped 方式的线连接，如果 `miterLimit < 相交部分的长度`，则会截断至线头（同斜截方式，相当于 `miterLimit = 1.0`）；如果 `miterLimit >= 相交部分的长度`，则绘制完整的相交部分。

但是对 LineJoinMiter 方式，如果 `miterLimit < 相交部分的长度`，则会截断至 `miterLimit` 所指定比例的长度；如果 `miterLimit >= 相交部分的长度`，则绘制完整的相交部分。

例如：

```
Graphics graph(pDC->m_hDC);
Pen redPen(Color::Red); // 画细线的红笔
Pen pen(Color::DarkGreen, 40.0f); // 画粗线的绿色笔
Point points[] = {Point(20, 100), Point(400, 130), Point(20, 160)}; // 点数组
pen.SetLineJoin(LineJoinMiter); // 斜接
//pen.SetLineJoin(LineJoinBevel); // 斜截
//pen.SetLineJoin(LineJoinRound); // 圆角
//pen.SetLineJoin(LineJoinMiterClipped); // 斜剪
//pen.SetMiterLimit(20.0f); // 设置斜接限长
graph.DrawLines(&pen, points, 3); // 画粗线
graph.DrawLines(&redPen, points, 3); // 画细线
```

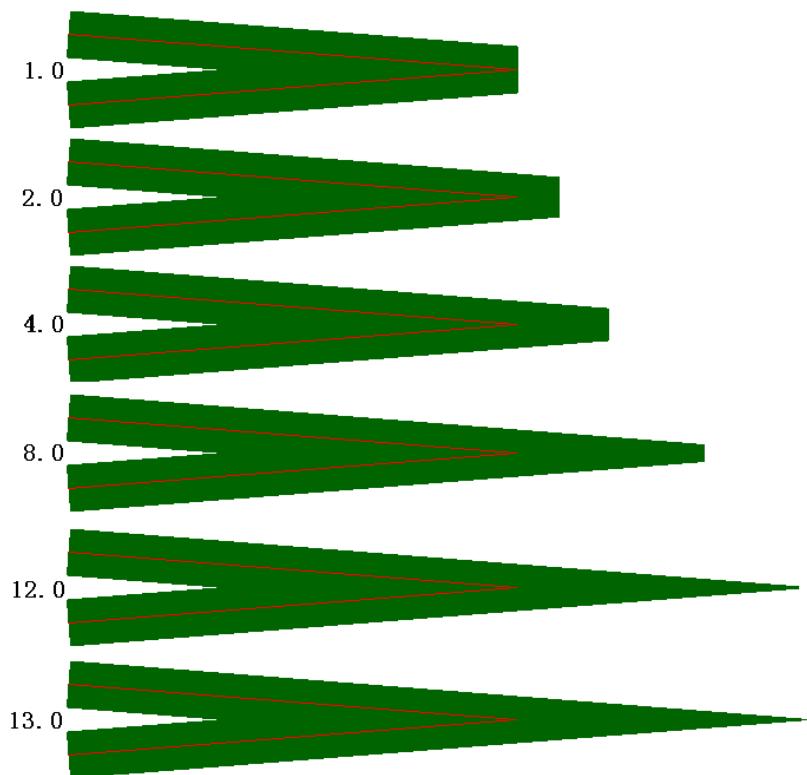
输出结果如：



如果不修改斜接线连接 LineJoinMiter 方式下的线长限制 (0.0f~13.0f)，则可得到不同截断长度的斜交角。例如：

```
pen.SetLineJoin(LineJoinMiter); // 斜接  
pen.SetMiterLimit(1.0f/*~13.0f*/); // 设置斜接限长
```

输出结果如：

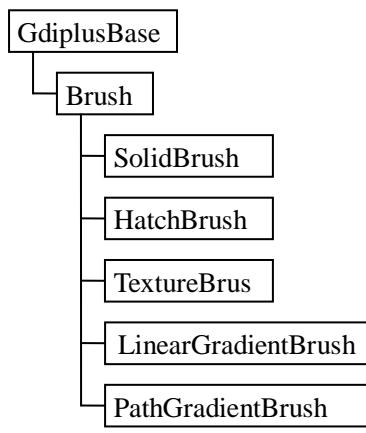


不同斜接限长下的斜接线连接 LineJoinMiter

4) Brush (刷)

与 GDI 中的一样，GDI+中的刷（brush 画刷/画笔）也是画填充图的工具，GDI+中也有与 GDI 相对应的实心刷（单色刷）、条纹刷（影线刷）和纹理刷（图像刷）。不过，GDI+又新增加了功能强大的线性渐变刷和路径渐变刷，而且还为所有这些刷各自建立了对应的类，基类是 Brush （功能少）。

下面是 GDI+中各种刷类的层次结构图：



GDI+刷类的层次结构

所有刷类都被定义在头文件 `GdiplusBrush.h` 中。

(0) 刷基类 Brush

`Brush` 是所有 GDI+具体刷类的基类，`Brush` 类没有自己的公用构造函数，属于非实例化类（用户不能创建 `Brush` 类的对象和实例），只是定义了三个公用的成员函数（接口）：

```
Brush *Clone(VOID) const; // 克隆，用于复制 Brush 及其派生类对象  
Status GetLastStatus(VOID); // 获取最后状态，返回刷对象最近的错误状态  
BrushType GetType(VOID); // 获取类型，返回当前（派生）刷的类型枚举常量
```

下面是 `BrushType` 枚举类型的定义：(`GdiplusEnums.h`)

```
typedef enum {  
    BrushTypeSolidColor = 0, // 实心单色刷  
    BrushTypeHatchFill = 1, // 影线条纹填充刷  
    BrushTypeTextureFill = 2, // 图像纹理填充刷  
    BrushTypePathGradient = 3, // 路径渐变刷  
    BrushTypeLinearGradient = 4 // 线性渐变刷  
} BrushType;
```

(1) 实心刷 SolidBrush

GDI+中，实心的单色刷对应于 `SolidBrush` 类，它只有一个构造函数：

`SolidBrush(const Color &color);`

输入参数为颜色对象的引用。

该类很简单，只有两个有关刷子颜色的成员函数：

```
Status GetColor(Color *color) const;
```

```
Status SetColor(const Color &color);
```

在前面的例子中，已经多次使用了 SolidBrush 类。下面就再举一个画正叶曲线的例子：

正叶曲线：(极坐标方程)

$$r = |l \cos \frac{n}{2} \theta|$$

其中， l 为叶片长度、 n 为叶片数目。

利用极坐标系到直角坐标系的转换公式：

$$\begin{cases} x = r \cos \theta \\ y = r \sin \theta \end{cases} \quad 0 \leq \theta \leq 2\pi$$

容易写出对应的程序。

因为 GDI+ 并没有画正叶曲线的专门函数，所以需要用多边形、样条曲线或图形路径来刻画它。可以使用填充多边形、填充封闭曲线和填充图形路径等方式来进行绘制。

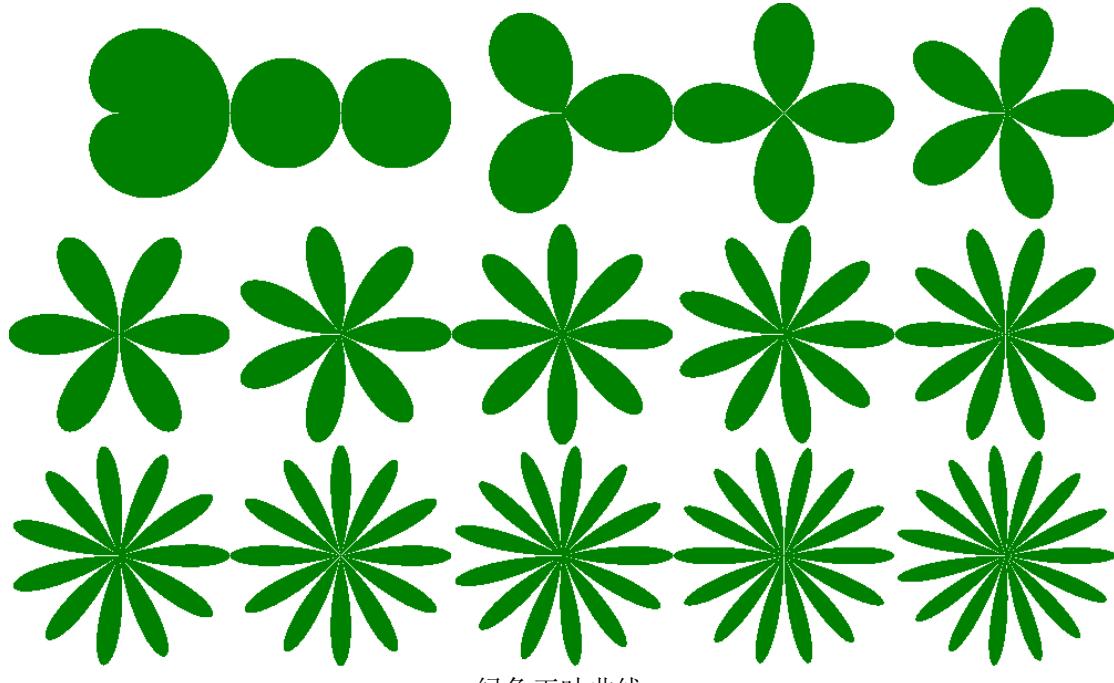
我现在改主意了，将这个例子留作作业。函数原型类似于：

```
void DrawLeaves(Graphics &graph, Color col, Point &O, int l, int n);
```

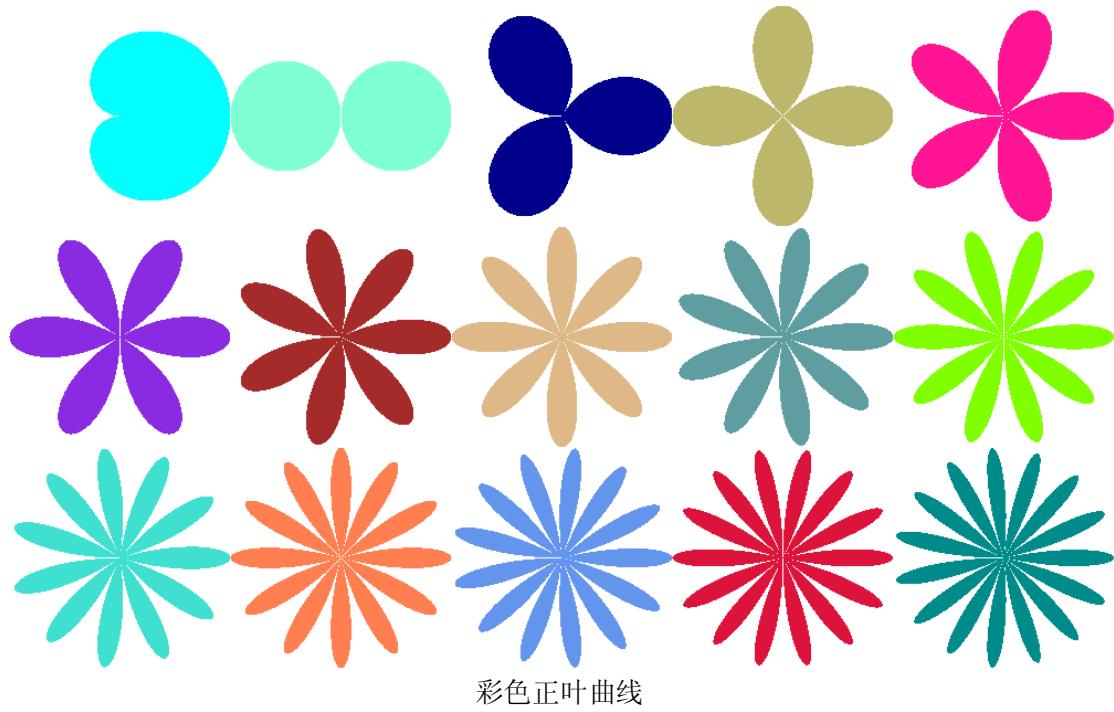
如果调用序列为：

```
Graphics graph(pDC->m_hDC);
Color cols[] = {Color::Aqua, Color::Aquamarine, Color::DarkBlue, Color::DarkKhaki,
    Color::DeepPink, Color::BlueViolet, Color::Brown, Color::BurlyWood,
    Color::CadetBlue, Color::Chartreuse, Color::Turquoise, Color::Coral,
    Color::CornflowerBlue, Color::Crimson, Color::DarkCyan};
bool color = true; // false;
for (int i = 0; i < 15; i++) DrawLeaves(graph, color ? cols[i] : Color::Green,
    Point(100 + 200*(i % 5), 100 + 200 * (i / 5)), 100, i + 1);
```

则输出结果为：



绿色正叶曲线



(2) 条纹刷 HatchBrush

条纹是一种重复填充的小方形图案，一般为横线、竖线、斜线和小方块等构成。

GDI+中，条纹刷（hatch brush 影线刷/阴影刷）对应于 HatchBrush 类，它也只有一个构造函数：

HatchBrush(HatchStyle hatchStyle, const Color &foreColor, const Color &backColor = Color());
其中：第一个参数为条纹类型，第二个参数为前景色（条纹色），第三个参数为背景色（空隙色）。

与构造函数的三个输入参数相对应，HatchBrush 类有三个成员函数：

```
HatchStyle GetHatchStyle(VOID) const;
Status GetForegroundColor(Color *color) const;
Status GetBackgroundColor(Color *color) const;
```

GDI+中一共有 53 种条纹风格，而 GDI 中只有 6 种。条纹风格枚举 HatchStyle 也被定义在头文件 GdipplusEnums.h 中：

```
enum HatchStyle {
    HatchStyleHorizontal,           // 0: 横线
    HatchStyleVertical,             // 1: 竖线
    HatchStyleForwardDiagonal,     // 2: 正斜线
    HatchStyleBackwardDiagonal,    // 3: 反斜线
    HatchStyleCross,                // 4: 十字线
    HatchStyleDiagonalCross,       // 5: 斜十字线
    HatchStyle05Percent,           // 6: 5%
    HatchStyle10Percent,            // 7: 10%
    HatchStyle20Percent,            // 8: 20%
    HatchStyle25Percent,            // 9: 25%
```

HatchStyle30Percent,	// 10: 30%
HatchStyle40Percent,	// 11: 40%
HatchStyle50Percent,	// 12: 50%
HatchStyle60Percent,	// 13: 60%
HatchStyle70Percent,	// 14: 70%
HatchStyle75Percent,	// 15: 75%
HatchStyle80Percent,	// 16: 80%
HatchStyle90Percent,	// 17: 90%
HatchStyleLightDownwardDiagonal,	// 18: 亮下斜
HatchStyleLightUpwardDiagonal,	// 19: 亮上斜
HatchStyleDarkDownwardDiagonal,	// 20: 暗下斜
HatchStyleDarkUpwardDiagonal,	// 21: 暗上斜
HatchStyleWideDownwardDiagonal,	// 22: 宽下斜
HatchStyleWideUpwardDiagonal,	// 23: 宽上斜
HatchStyleLightVertical,	// 24: 亮竖线
HatchStyleLightHorizontal,	// 25: 亮横线
HatchStyleNarrowVertical,	// 26: 窄竖线
HatchStyleNarrowHorizontal,	// 27: 窄横线
HatchStyleDarkVertical,	// 28: 暗竖线
HatchStyleDarkHorizontal,	// 29: 暗横线
HatchStyleDashedDownwardDiagonal,	// 30: 虚下斜
HatchStyleDashedUpwardDiagonal,	// 31: 虚上斜
HatchStyleDashedHorizontal,	// 32: 虚横线
HatchStyleDashedVertical,	// 33: 虚竖线
HatchStyleSmallConfetti,	// 34: 小纸屑
HatchStyleLargeConfetti,	// 35: 大纸屑
HatchStyleZigZag,	// 36: 锯齿
HatchStyleWave,	// 37: 波形
HatchStyleDiagonalBrick,	// 38: 斜砖
HatchStyleHorizontalBrick,	// 39: 横砖
HatchStyleWeave,	// 40: 编织
HatchStylePlaid,	// 41: 格子
HatchStyleDivot,	// 42: 草皮
HatchStyleDottedGrid,	// 43: 点线网格
HatchStyleDottedDiamond,	// 44: 斜点线
HatchStyleShingle,	// 45: 鹅卵石
HatchStyleTrellis,	// 46: 细格
HatchStyleSphere,	// 47: 球面
HatchStyleSmallGrid,	// 48: 小网格
HatchStyleSmallCheckerBoard,	// 49: 小跳棋盘
HatchStyleLargeCheckerBoard,	// 50: 大跳棋盘
HatchStyleOutlinedDiamond,	// 51: 斜纲线
HatchStyleSolidDiamond,	// 52: 实菱形

```

HatchStyleTotal, // = 53 (0 ~ 52): 条纹风格总数
HatchStyleLargeGrid = HatchStyleCross, // 4: 大网格

HatchStyleMin      = HatchStyleHorizontal, // 0: 条纹风格最小值
HatchStyleMax      = HatchStyleTotal - 1, // 52: 条纹风格最大值
};

例如:

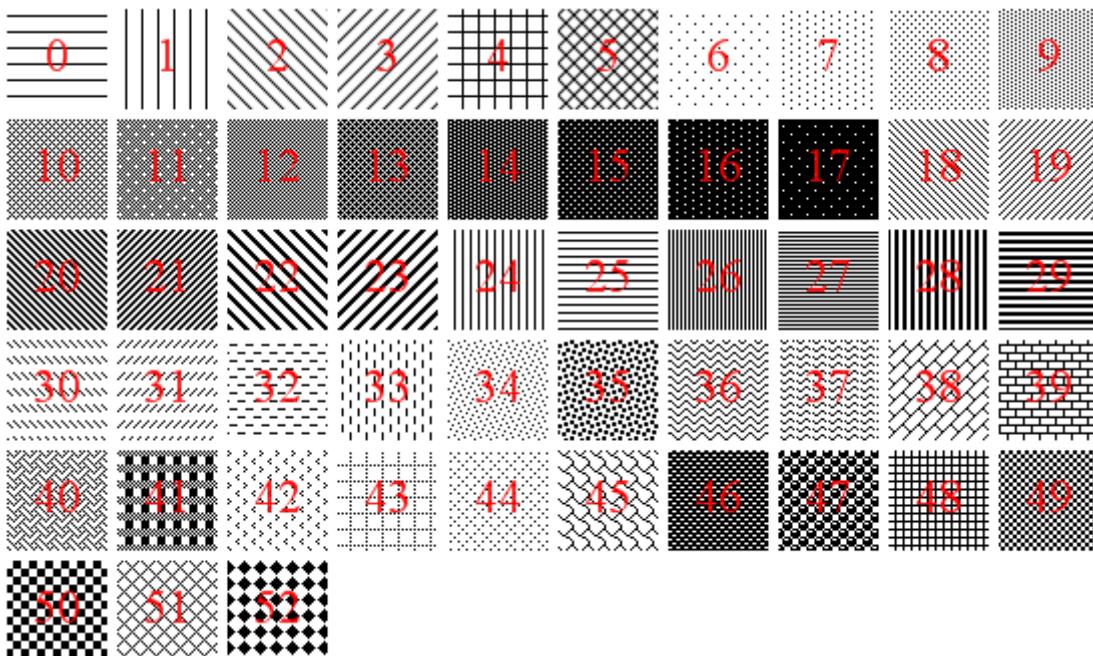
```

```

Graphics graph(pDC->m_hDC);
Pen pen(Color::Black);
SolidBrush textBrush(Color::Red);
FontFamily fontFamily(L"Times New Roman");
Font font(&fontFamily, 18);
CString str;
StringFormat sfmt; // 文本格式
sfmt.SetAlignment(StringAlignmentCenter); // 水平对齐
sfmt.SetLineAlignment(StringAlignmentCenter); // 垂直对齐
int w = 50, h = 50, s = 5;
for (int i = 0; i < 53; i++) { // 主循环
    HatchBrush brush(HatchStyle(i), Color::Black, Color::White);
    RectF rect(REAL(s + (i % 10) * (w + s)),
               REAL(s + (i / 10) * (h + s)), REAL(w), REAL(h));
    graph.FillRectangle(&brush, rect); // 画条纹块
    str.Format(L"%d", i); // 绘制数字编号的文本串:
    graph.DrawString(str, str.GetLength(), &font, rect, &fmt, &textBrush);
}

```

输出结果为:



条纹刷的条纹风格

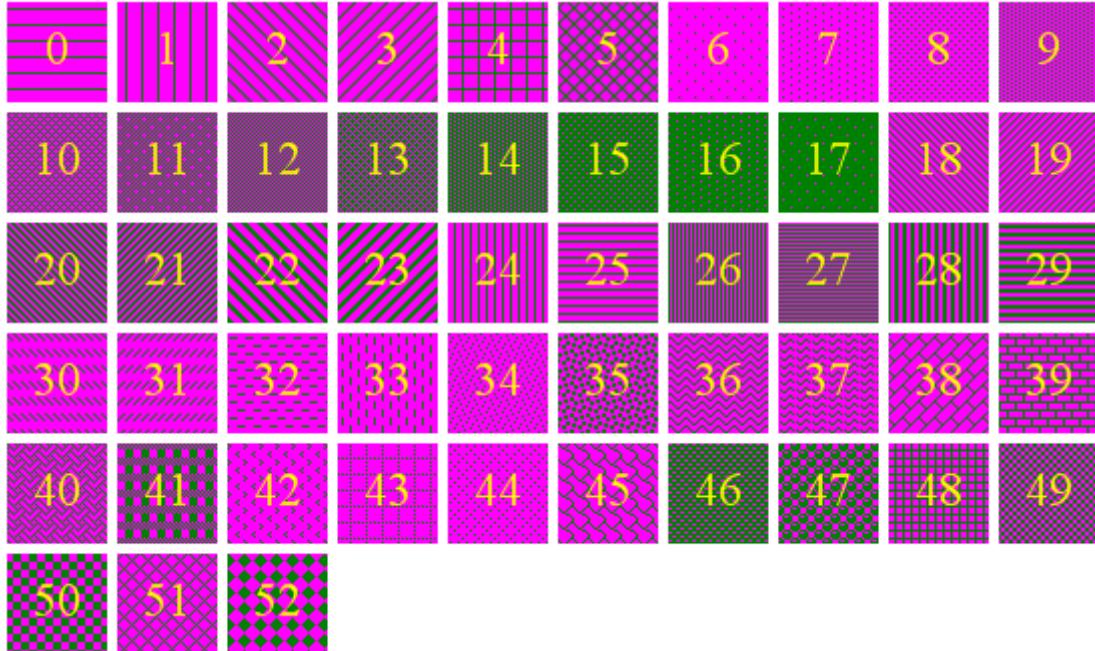
也可以改变条纹刷的前景和背景色。例如修改上例中的如下两条语句：

```
SolidBrush textBrush(Color::Yellow);
```

和

```
HatchBrush brush((HatchStyle)i, Color::Green, Color::Magenta);
```

则输出结果变为：



条纹刷的条纹风格与前背景色

与 GDI一样，在GDI+中也可以调整条纹刷和图像刷的起点。这需要使用图像类 Graphics 的成员函数

```
Status SetRenderingOrigin(INT x, INT y);
```

来设置渲染原点为(x, y)，缺省为(0, 0)。

在 GDI 中，使用的是 CDC 类的成员函数 SetBrushOrg 来设置刷的原点：

```
CPoint SetBrushOrg( int x, int y );
```

```
CPoint SetBrushOrg( POINT point );
```

(3) 纹理刷 TextureBrush

纹理刷 (texture brush) 就是图像刷，它将刷中所装入的图像，在目标区域中进行平铺，可达到纹理效果。

GDI 中也有图像刷，但仅限于使用位图资源和（非常费事才能使用）BMP 文件。

在 GDI+中，纹理刷所对应的是 TextureBrush 类，它有 7 个构造函数：

```
TextureBrush(Image* image, WrapMode wrapMode = WrapModeTile);
```

```
TextureBrush(Image* image, WrapMode wrapMode, const RectF &dstRect);
```

```
TextureBrush(Image *image, const RectF &dstRect, const ImageAttributes  
*imageAttributes = NULL);
```

```
TextureBrush(Image *image, const Rect &dstRect, const ImageAttributes  
*imageAttributes = NULL);
```

```
TextureBrush(Image* image, WrapMode wrapMode, const Rect &dstRect);
```

```

TextureBrush(Image* image, WrapMode wrapMode, REAL dstX, REAL dstY, REAL
dstWidth, REAL dstHeight);
TextureBrush(Image* image, WrapMode wrapMode, N INT dstX, INT dstY, INT
dstWidth, INT dstHeight);

```

其中，最常用的是第一个。它的第一个参数是图像对象的指针，第二个参数是排列方式的枚举常量：(GdiplusEnums.h)

```

typedef enum {
    WrapModeTile = 0, // 平铺（瓦）（缺省值）
    WrapModeTileFlipX = 1, // 平铺且 X 向翻转（相邻列左右翻转）
    WrapModeTileFlipY = 2, // 平铺且 Y 向翻转（相邻行上下翻转）
    WrapModeTileFlipXY = 3, // 平铺且 XY 向翻转（相邻行列左右上下翻转）
    WrapModeClamp = 4 // 不平铺（不重复，夹住）
} WrapMode;

```

还可以用纹理刷类的成员函数：

```

Status SetWrapMode(WrapMode wrapMode);
WrapMode GetWrapMode() const;

```

来设置和获取刷的排列方式。

例如：

```

Graphics graph(pDC->m_hDC);
Image img(L"张东健.bmp");
TextureBrush brush(&img, WrapModeTile/*FlipXY*/); // WrapModeClamp
RECT rect;
GetClientRect(&rect);
graph.FillRectangle(&brush, RectF(0.0f, 0.0f, REAL(rect.right), REAL(rect.bottom)));

```

输出结果如：



平铺 WrapModeTile



平铺且 X 向翻转 WrapModeTileFlipX



平铺且 Y 向翻转 WrapModeTileFlipY 平铺且 XY 向翻转 WrapModeTileFlipXY



不平铺 WrapModeClamp

纹理刷排列方式

纹理刷类 TextureBrush 中, 还有几个成员函数, 可以对刷中的图像进行平移(translate)、旋转 (rotate) 和缩放 (scale) 等变换 (transform): (这是 GDI 里所没有的功能)

```
Status TranslateTransform(REAL dx, REAL dy, MatrixOrder order = MatrixOrderPrepend);
```

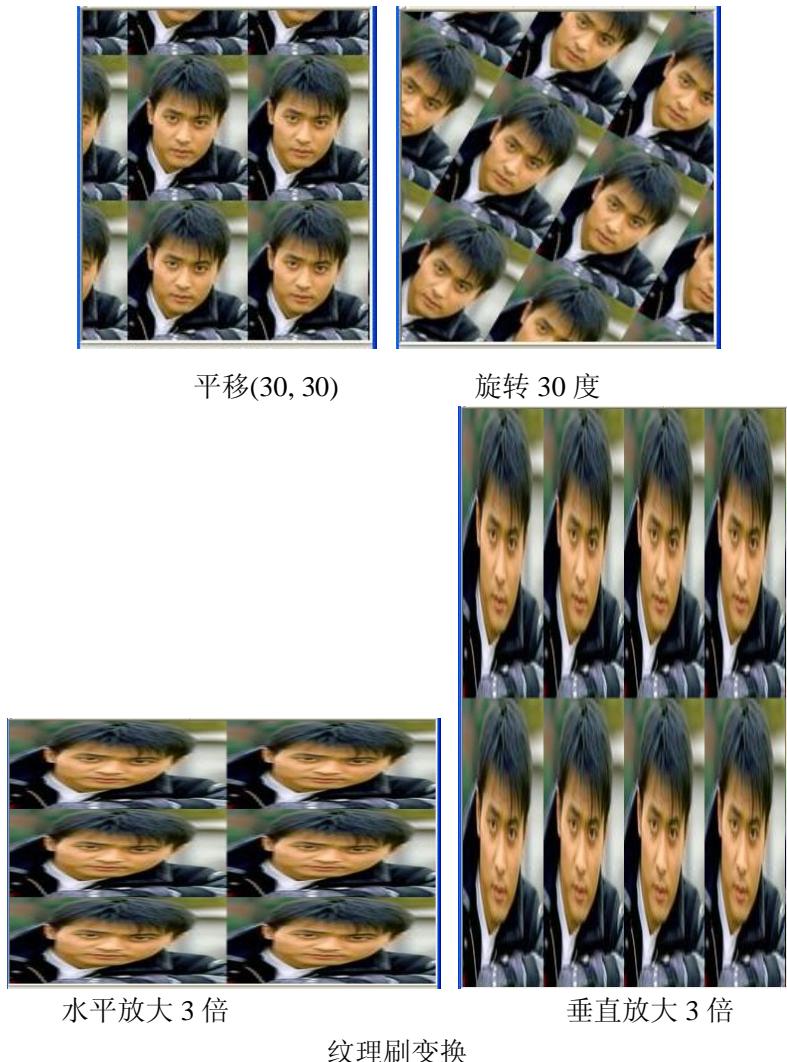
```
Status RotateTransform(REAL angle, MatrixOrder order = MatrixOrderPrepend);
```

```
Status ScaleTransform(REAL sx, REAL sy, MatrixOrder order = MatrixOrderPrepend);
```

例如:

```
Graphics graph(pDC->m_hDC);
Image img(L"张东健.bmp");
TextureBrush brush(&img);
//brush.TranslateTransform(30, 30); // 平移(30, 30)
brush.RotateTransform(30); // 旋转 30 度
//brush.ScaleTransform(3, 1); // 水平放大 3 倍
//brush.ScaleTransform(1, 3); // 垂直放大 3 倍
RECT rect;
GetClientRect(&rect);
```

```
graph.FillRectangle(&brush, RectF(0.0f, 0.0f, REAL(rect.right), REAL(rect.bottom)));  
输出结果为：
```



(4) 线性渐变刷 **LinearGradientBrush**

线性渐变刷（linear gradient brush 线性梯度刷）使用逐渐变化的颜色填充目标区域。是 GDI+新增的功能。

线性渐变刷所对应的类为 **LinearGradientBrush**，它有 6 个构造函数：

```
LinearGradientBrush(const Point& point1, const Point& point2, const Color& color1,  
const Color& color2);  
LinearGradientBrush(const Rect& rect, const Color& color1, const Color& color2,  
LinearGradientMode mode);  
LinearGradientBrush(const Rect& rect, const Color& color1, const Color& color2,  
REAL angle, BOOL isAngleScalable = FALSE);  
LinearGradientBrush(const PointF& point1, const PointF& point2, const Color& color1,  
const Color& color2);  
LinearGradientBrush(const RectF& rect, const Color& color1, const Color& color2,  
LinearGradientMode mode);
```

```
LinearGradientBrush(const RectF& rect, const Color& color1, const Color& color2,  
REAL angle, BOOL isAngleScalable = FALSE);
```

前三个是整数版，后三个是对应的浮点数版。三种构造函数中，第一个是点到点、第二个是矩形与渐变模式、第三个是矩形与旋转角度。

● 点到点渐变

其中最常用的是第一个：

```
LinearGradientBrush(p1, p2, col1, col2);
```

这里的渐变是指刷子所填充的颜色，从 col1 连续变化到 col2。

若 p1 和 p2 点的 y 值相等，则为水平方向的渐变；若 p1 和 p2 点的 x 值相等，则为垂直方向的渐变；p1 和 p2 点的 x 和 y 值都不相等，则为斜对角方向的渐变。例如：

```
Graphics graph(pDC->m_hDC);  
Point p1(10, 10), p2(110, 10), p3(10, 110), p4(230, 10), p5(330, 110);  
Size size(100, 100);  
Color col1(255, 0, 0), col2(0, 0, 255);  
LinearGradientBrush hbrush(p1, p2, col1, col2);  
graph.FillRectangle(&hbrush, Rect(p1, size));  
LinearGradientBrush vbrush(p1, p3, col1, col2);  
graph.FillRectangle(&vbrush, Rect(Point(120, 10), size));  
LinearGradientBrush dbrush(p4, p5, col1, col2);  
graph.FillRectangle(&dbrush, Rect(Point(230, 10), size));
```



水平渐变

垂直渐变

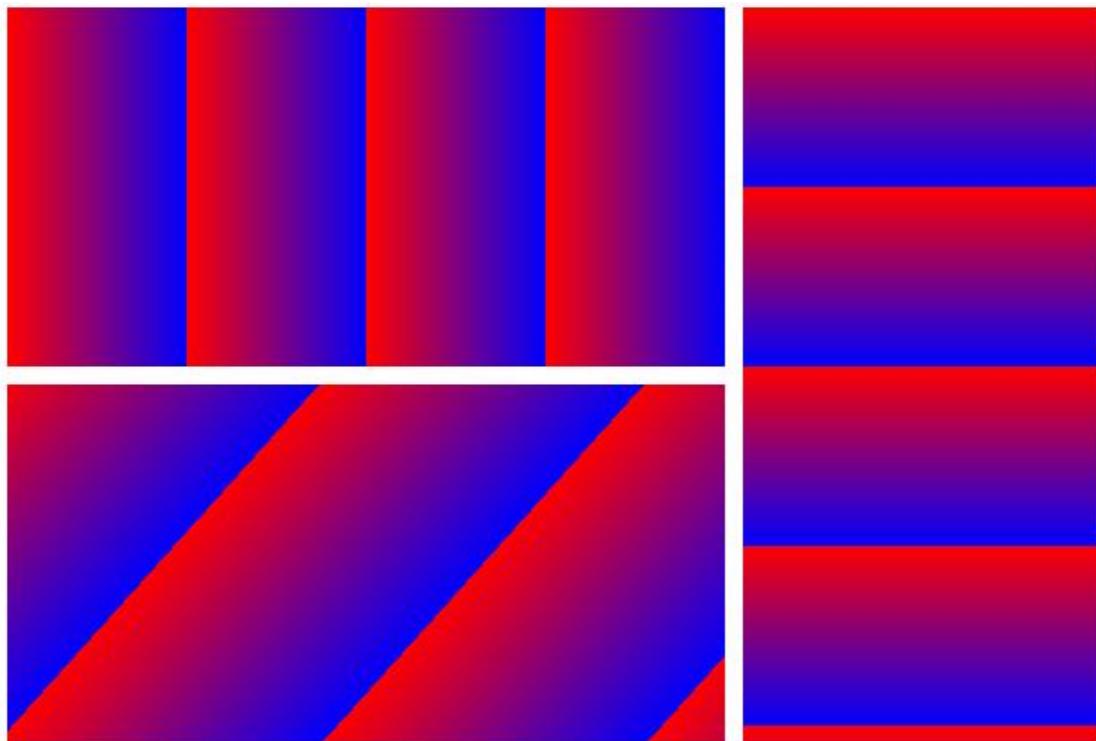
对角渐变

线性渐变刷

其实，线性渐变刷缺省是按 WrapModeTile 平铺方式重复排列的（原点是 point1 或 rect 的左上角），例如：

```
Graphics graph(pDC->m_hDC);  
Point p1(10, 10), p2(110, 10), p3(10, 110);  
Color col1(255, 0, 0), col2(0, 0, 255);  
LinearGradientBrush hbrush(p1, p2, col1, col2);  
//hbrush.SetWrapMode(WrapModeTileFlipX);  
graph.FillRectangle(&hbrush, Rect(p1, Size(400, 200)));  
LinearGradientBrush vbrush(p1, p3, col1, col2);  
//vbrush.SetWrapMode(WrapModeTileFlipX);  
graph.FillRectangle(&vbrush, Rect(Point(420, 10), Size(200, 410)));  
LinearGradientBrush dbrush(p1, Point(110, 100), col1, col2);  
//dbrush.SetWrapMode(WrapModeTileFlipX);  
graph.FillRectangle(&dbrush, Rect(Point(10, 220), Size(400, 200)));
```

则输出结果为：



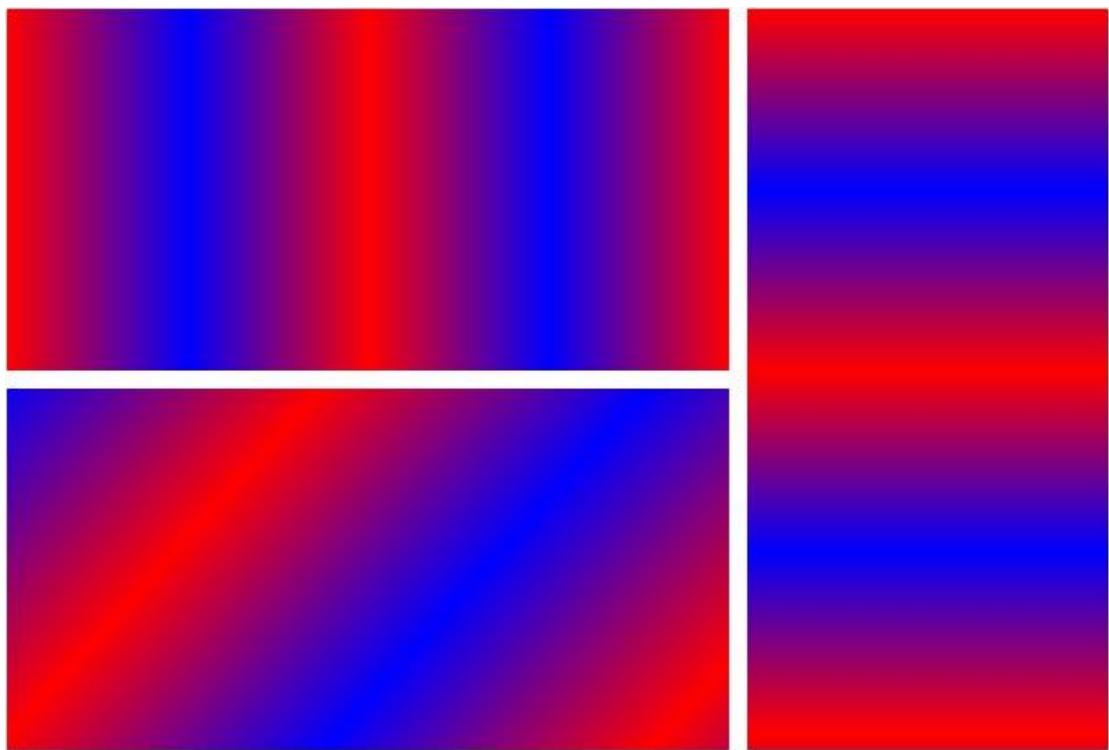
按平铺重复排列的线性渐变刷

你也可以将上面代码中的注释符“//”去掉，利用线性渐变刷类的成员函数

```
Status SetWrapMode(WrapMode wrapMode);
```

来设置画刷的排列方式为 WrapModeTileFlipX 平铺并水平翻转（我做过测试，用垂直翻转 WrapModeTileFlipY 无效，但可以改用水平垂直翻转 WrapModeTileFlipXY）。

输出结果为：



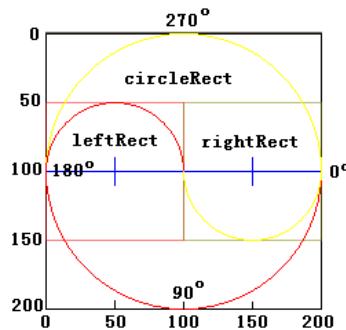
按平铺加水平翻转重复排列的线性渐变刷

下面这个例子，是利用水平线性渐变刷来画阴阳八卦中的阴阳鱼：

```

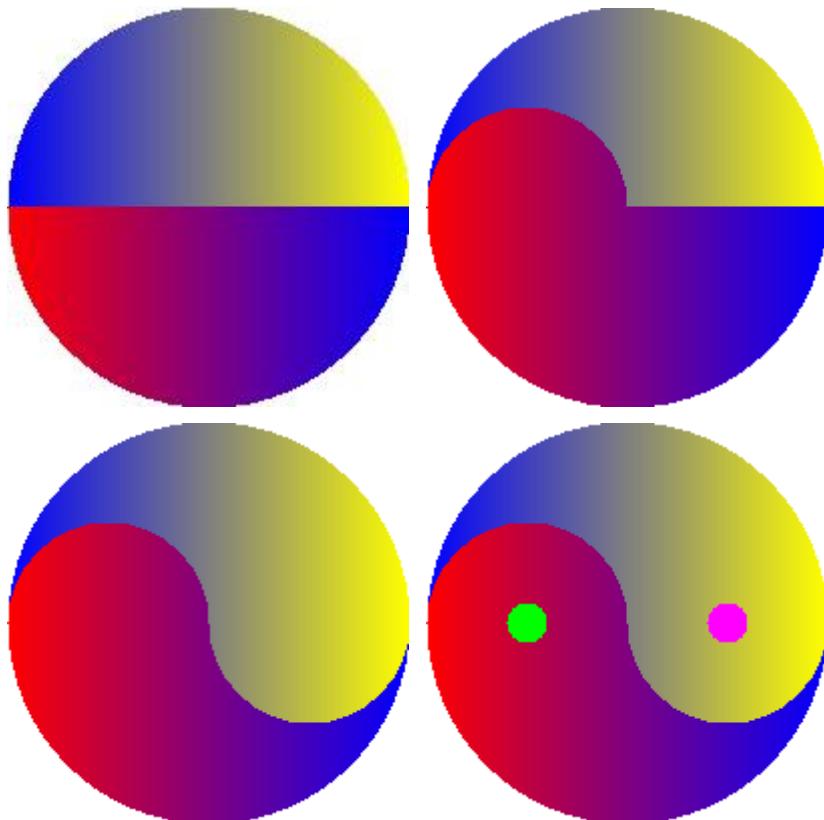
LinearGradientBrush R2BBrush(Point(0, 10), Point(200, 10), Color(255, 0, 0), Color(0, 0, 255));
LinearGradientBrush B2YBrush(Point(0, 10), Point(200, 10), Color(0, 0, 255), Color(255, 255, 0));
Pen bluePen(Color(255, 0, 0, 255));
Rect circleRect(0, 0, 200, 200);
Rect leftRect(0, 50, 100, 100);
Rect rightRect(100, 50, 100, 100);
Graphics graph(pDC->m_hDC);
graph.FillPie(&R2BBrush, circleRect, 0.0f, 180.0f);
graph.FillPie(&B2YBrush, circleRect, 180.0f, 180.0f);
graph.FillPie(&R2BBrush, leftRect, 180.0f, 180.0f);
graph.FillPie(&B2YBrush, rightRect, 0.0f, 180.0f);
int r = 10;
graph.FillEllipse(new SolidBrush(Color(0, 255, 0)), 50 - r, 100 - r, 2 * r, 2 * r);
graph.FillEllipse(new SolidBrush(Color(255, 0, 255)), 150 - r, 100 - r, 2 * r, 2 * r);

```



阴阳鱼例程中参数的含义

分步的输出结果为：



阴阳鱼

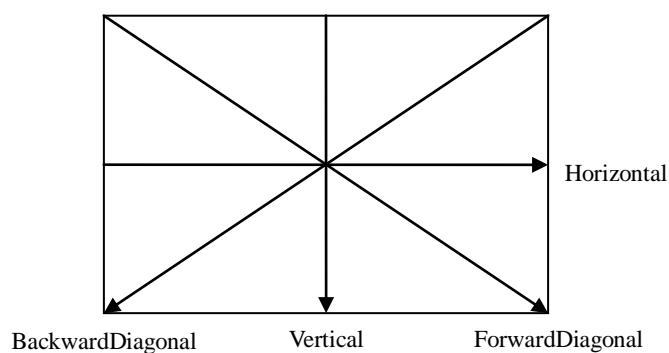
● 矩形渐变模式

第二个构造函数

```
LinearGradientBrush(const Rect& rect, const Color& color1, const Color& color2,  
LinearGradientMode mode);
```

的第一个输入参数是确定变换范围的矩形，最后一个输入参数，是一个线性渐变模式枚举常量：

```
typedef enum {  
    LinearGradientModeHorizontal = 0, // 水平渐变  
    LinearGradientModeVertical = 1, // 垂直渐变  
    LinearGradientModeForwardDiagonal = 2, // 正对角渐变  
    LinearGradientModeBackwardDiagonal = 3 // 反对角渐变  
} LinearGradientMode;
```

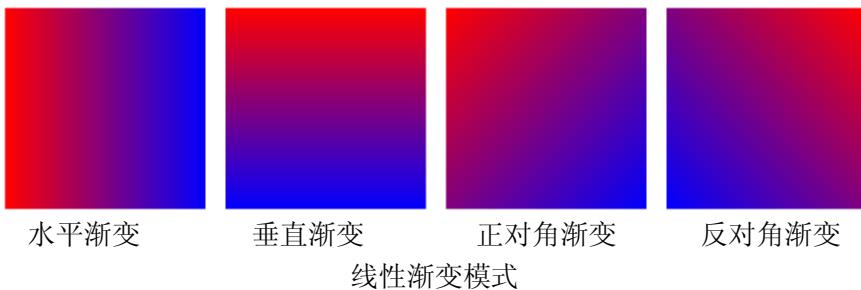


□ 矩形线性渐变模式的含义

例如：

```
Graphics graph(pDC->m_hDC);
Color col1(255, 0, 0), col2(0, 0, 255);
Size size(100, 100);
Rect rect1(Point(10, 10), size);
LinearGradientBrush hbrush(rect1, col1, col2, LinearGradientModeHorizontal);
graph.FillRectangle(&hbrush, rect1);
Rect rect2(Point(120, 10), size);
LinearGradientBrush vbrush(rect2, col1, col2, LinearGradientModeVertical);
graph.FillRectangle(&vbrush, rect2);
Rect rect3(Point(230, 10), size);
LinearGradientBrush fbrush(rect3, col1, col2, LinearGradientModeForwardDiagonal);
graph.FillRectangle(&fbrush, rect3);
Rect rect4(Point(340, 10), size);
LinearGradientBrush bbrush(rect4, col1, col2, LinearGradientModeBackwardDiagonal);
graph.FillRectangle(&bbrush, rect4);
```

输出结果为：



● 旋转角及伸缩角渐变

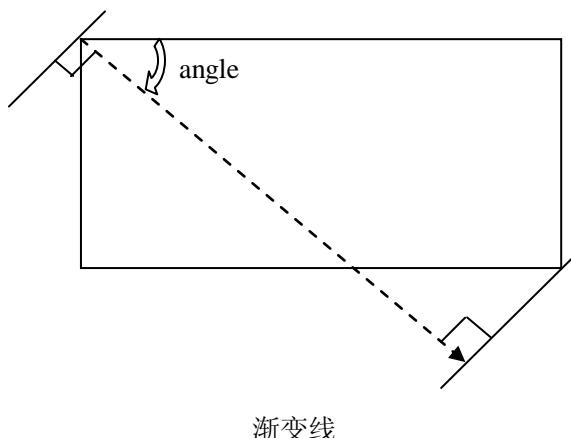
第三个线性渐变刷的构造函数为

```
LinearGradientBrush(const Rect& rect, const Color& color1, const Color& color2,
REAL angle, BOOL isAngleScalable = FALSE);
```

刷的渐变方向和基本范围（渐变线），由矩形起点和最后两个参数（转角、缩放）来确定。

isAngleScalable = FALSE (转角不可伸缩)

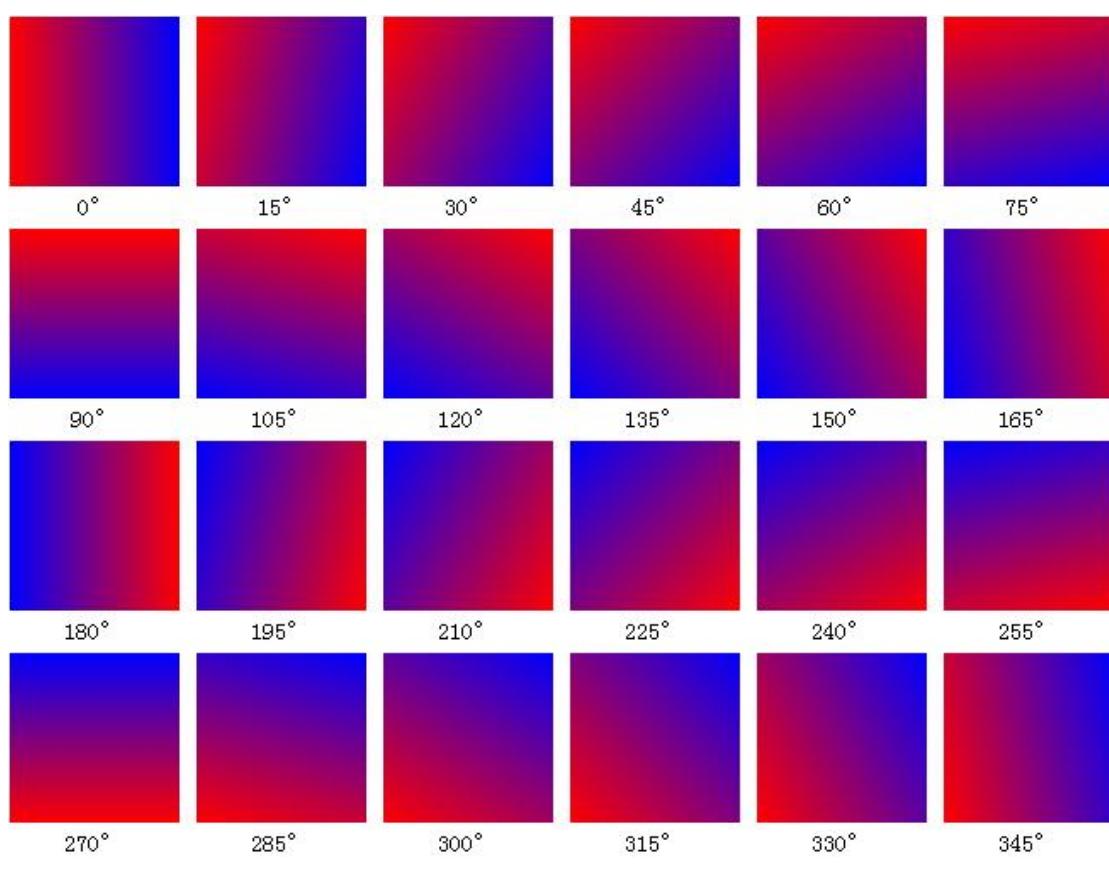
则 angle 表示渐变方向的角度（相对于矩形起点的水平线，顺时针为正向），渐变线为：



例如：

```
Graphics graph(pDC->m_hDC);
Pen pen(Color::Black);
SolidBrush textBrush(Color::Black);
FontFamily fontFamily(L"宋体");
Font font(&fontFamily, 10.5);
wchar_t str[10];
StringFormat sfmt;
sfmt.SetAlignment(StringAlignmentCenter);
Color col1(255, 0, 0), col2(0, 0, 255);
int l = 100, d = 10, ld = l + d, sd = 15;
Size size(l, l);
for (int i = 0; i < 24; i++) {
    Rect rect(Point(d + (i % 6) * ld, d + (i / 6) * (ld + sd)), size);
    REAL angle = i * 15.0f;
    LinearGradientBrush brush(rect, col1, col2, angle);
    graph.FillRectangle(&brush, rect);
    swprintf_s(str, 10, L"%g °", angle);
    graph.DrawString(str, INT(wcslen(str)), &font, PointF(rect.X + rect.Width / 2.0f,
        rect.GetBottom() + 5.0f), &sfmt, &textBrush);
}
```

输出结果为：



线性渐变刷的旋转角度

isAngleScalable = TRUE (转角可伸缩)

则渐变线的方向角 β 由 $\beta = \arctan(\text{width} / \text{height}) \tan(\phi)$ 来共同确定：

$$\beta = \arctan(\text{width} / \text{height}) \tan(\phi)$$

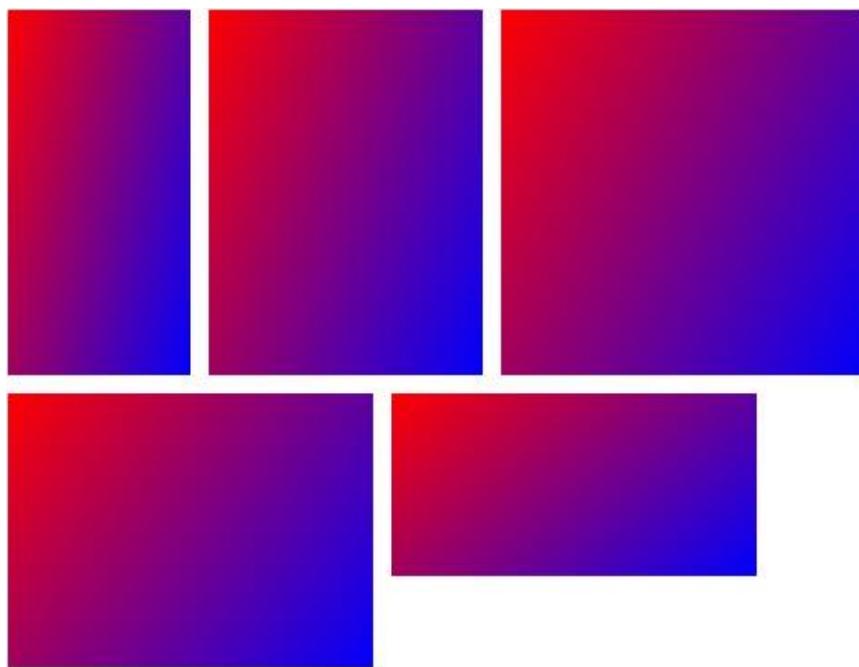
可见，对正方形， $\text{width} = \text{height}$ ，所以 $\beta = \phi$ ；对扁矩形， $\text{width} > \text{height}$ ，则 $\beta > \phi$ ；但对高矩形， $\text{width} < \text{height}$ ，则 $\beta < \phi$ 。

另外，要注意，因为计算过程中，用到了正切函数，所以 $-90^\circ < \phi < 90^\circ$ 才有意义。

例如：

```
Graphics graph(pDC->m_hDC);
REAL angle = 30.0f;
Color col1(255, 0, 0), col2(0, 0, 255);
Rect rect1(Point(10, 10), Size(100, 200));
LinearGradientBrush brush1(rect1, col1, col2, angle, TRUE);
graph.FillRectangle(&brush1, rect1);
Rect rect2(Point(120, 10), Size(150, 200));
LinearGradientBrush brush2(rect2, col1, col2, angle, TRUE);
graph.FillRectangle(&brush2, rect2);
Rect rect3(Point(280, 10), Size(200, 200));
LinearGradientBrush brush3(rect3, col1, col2, angle, TRUE);
graph.FillRectangle(&brush3, rect3);
Rect rect4(Point(10, 220), Size(200, 150));
LinearGradientBrush brush4(rect4, col1, col2, angle, TRUE);
graph.FillRectangle(&brush4, rect4);
Rect rect5(Point(220, 220), Size(200, 100));
LinearGradientBrush brush5(rect5, col1, col2, angle, TRUE);
graph.FillRectangle(&brush5, rect5);
```

输出结果为：（可见只要转角相同，虽然不同长宽比下的渐变渐变的方向不同，但矩形的渐变效果却很类似，这正是设计伸缩角渐变的目的所在）



转角相同 (30°) 长宽比不同矩形的角度缩放线性渐变刷

● 多色渐变

线性渐变刷还有很多其他功能，如可利用刷的成员函数：

```
Status SetInterpolationColors(const Color *presetColors, const REAL *blendPositions, INT count);
```

来设置多色渐变。其中，`presetColors` 为多色数组、`blendPositions` 为以百分比表示的对应混色点的位置（首、尾值必须为 0.0f 和 1.0f，中间的值应该按递增序排列）、`count` 为颜色和混色点位的数目。

例如：

```
Color cols[] = {Color::Red, Color::Orange, Color::Yellow, Color::Green,
                 Color::Cyan, Color::Blue, Color::Purple, Color::Magenta};
REAL bps[] = {0.0f, 0.15f, 0.3f, 0.45f, 0.6f, 0.75f, 0.875f, 1.0f};
LinearGradientBrush brush(Point(10, 10), Point(810, 10), Color::Black, Color::White);
brush.SetInterpolationColors(cols, bps, 8);
graph.FillRectangle(&brush, Rect(10, 10, 800, 100));
```

输出结果为：



多色渐变

● 混色因子

缺省的线性渐变刷，是采用线性插值的方法，从起点色完全变换到终点色。但是，我们也可以刷的用成员函数：

```
Status SetBlend(const REAL *blendFactors, const REAL *blendPositions, INT count);
Status SetBlendBellShape(REAL focus, REAL scale);
Status SetBlendTriangularShape(REAL focus, REAL scale);
```

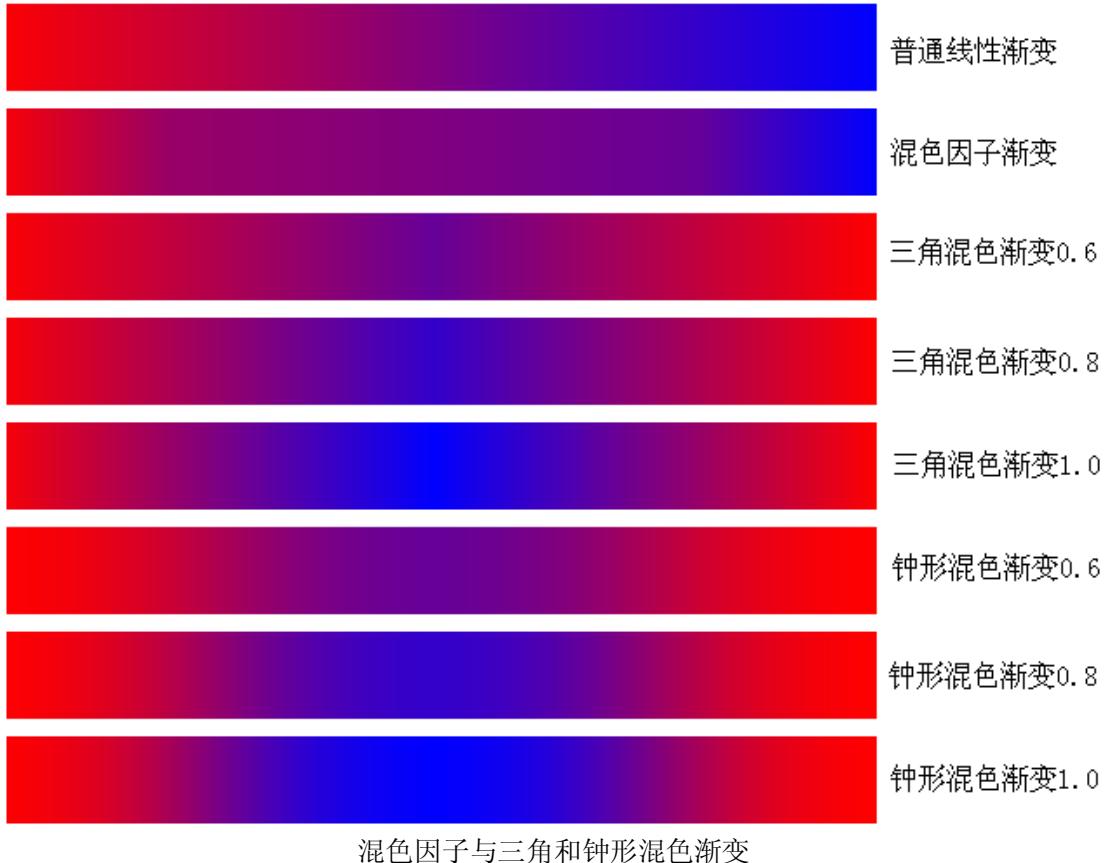
来改变这种渐变的默认行为。

例如：

```
Graphics graph(pDC->m_hDC);
REAL factors[4] = {0.0f, 0.4f, 0.6f, 1.0f};
REAL positions[4] = {0.0f, 0.2f, 0.8f, 1.0f};
LinearGradientBrush brush(Point(0, 0), Point(510, 0), Color::Red, Color::Blue);
graph.FillRectangle(&brush, 10, 10, 500, 50); // 普通
brush.SetBlend(factors, positions, 4); // 混色因子
graph.FillRectangle(&brush, 10, 70, 500, 50);
brush.SetBlendTriangularShape(0.5f, 0.6f); // 三角形 0.6
graph.FillRectangle(&brush, 10, 130, 500, 50);
brush.SetBlendTriangularShape(0.5f, 0.8f); // 三角形 0.8
graph.FillRectangle(&brush, 10, 190, 500, 50);
brush.SetBlendTriangularShape(0.5f, 1.0f); // 三角形 1.0
graph.FillRectangle(&brush, 10, 250, 500, 50);
```

```
brush.SetBlendBellShape(0.5f, 0.6f); // 钟形 0.6  
graph.FillRectangle(&brush, 10, 310, 500, 50);  
brush.SetBlendBellShape(0.5f, 0.8f); // 钟形 0.8  
graph.FillRectangle(&brush, 10, 370, 500, 50);  
brush.SetBlendBellShape(0.5f, 1.0f); // 钟形 1.0  
graph.FillRectangle(&brush, 10, 430, 500, 50);
```

输出结果为：



另外，也可以像纹理刷和条纹刷一样，设置线性渐变刷的渲染原点等。

路径渐变刷的内容，安排到本节的第 4 小节“4. 路径”中，在介绍过路径的基本概念和使用方法之后再来讲解。

3. 文字

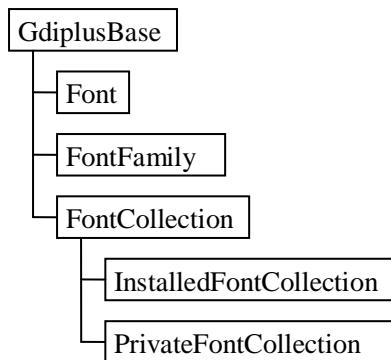
GDI+的文本排版和字体处理的功能比 GDI 的更加强大。特别是 Windows XP 提供了对 LCD（液晶）显示器的特殊优化功能，GDI+也提供了对应的 ClearType（清晰活字）文字处理技术，以增强字体的清晰度。另外，GDI+还提供了构造专用字体集的功能，可以包含私有的临时字体（不需预先安装到系统中）。

Windows 中使用的字体，一般是 TrueType（真实活字）字体（TTF = TrueType Font），它是 1991 年 Apple 和 Microsoft 联合开发的一种字体技术，采用二次 B 样条曲线来描述字符的轮廓。

在 GDI+中，与文字相关的类有：字体族类 `FontFamily`、字体类 `Font` 和字体集类

FontCollection 及其两个派生类 InstalledFontCollection（已安装字体集）和 PrivateFontCollection（专用字体集）。(在 GDI 中只有 CFont 一个字体类)

这些类的层次结构为：



1) 字体

(1) 字体族 **FontFamily**

字体族 (font family) 是一组具有同一种字形 (typeface)，但是风格 (style) 不同的字体 (font)。

其中，字形是指字体的种类，如 Arial、Times New Roman、宋体、楷体_GB2312。风格是指：正常 (regular)、粗体 (bold)、斜体 (italic)、粗斜体 (bold and italic)、下划线 (underline)、删除线 (strikeout) 等。

● 构造函数

字体族类 **FontFamily** 有两个构造函数：

```
FontFamily(); // 构造一个空字体族 (少用)  
FontFamily(const WCHAR *name, const FontCollection *fontCollection = NULL);  
// 构造具有指定名称 name，位于指定字体集 fontCollection 中的字体族
```

只要不是使用专用字体集中的字体，一般不需要设置第二个输入参数，取缺省的 NULL 即可。例如：

```
FontFamily fontFamily(L"宋体"); 或  
FontFamily fontFamily(L"Times New Roman");
```

● 显示当前系统已装入的字体（族）名称

可先利用（字体集 **FontCollection** 的派生类）已安装字体集类 **InstalledFontCollection** 的成员函数：

```
INT GetFamilyCount() const;  
Status GetFamilies(INT numSought, FontFamily *gpfamilies, INT *numFound) const;
```

来获取当前系统中已经安装了的字体集中的字体族的数目和对象指针。

然后再利用字体族类的成员函数

```
Status GetFamilyName(WCHAR name[LF_FACESIZE],  
WCHAR language = LANG_NEUTRAL) const;
```

来获取每个字体族的名称。其中 **LANG_NEUTRAL** 表示采用中立语言，即用户的缺省语言。

例如：(可创建一个带滚动视图类的单文档 MFC 应用程序 Fonts, 添加对 GDI+的支持)

```
void CFontsView::OnDraw(CDC* pDC) {  
    .....  
    InstalledFontCollection ifc;  
    int n = ifc.GetFamilyCount();  
    FontFamily *ffs = new FontFamily[n];  
    int found;  
    ifc.GetFamilies(n, ffs, &found);  
    wchar_t name[LF_FACESIZE];  
    Font font(L"宋体", 18);  
    SolidBrush textBrush(Color::Black);  
    Graphics graph(pDC->m_hDC);  
    wchar_t str[40];  
    swprintf_s(str, 40, L"当前系统中, 总共装有如下%d 种字体: ", n);  
    graph.DrawString(str, INT(wcslen(str)), &font, PointF(10.0f, 10.0f), &textBrush);  
    for (int i = 0; i < n; i++) {  
        ffs[i].GetFamilyName(name);  
        graph.DrawString(name, INT(wcslen(name)), &font,  
                         PointF(10.0f, 80.0f + 40 * i), &textBrush);  
        graph.DrawString(L"Font Family 字体族", 15, &Font(name, 18),  
                         PointF(300.0f, 80.0f + 40 * i), &textBrush);  
    }  
}
```

输出结果如：



在我的机器中一共装有 177 种字体，其中 161 种是西文字体，剩下的 16 种为中文字体。它们大都是 Windows XP 和 Office 2003 装入的，也有一些是 Adobe Acrobat 7.0 等其他软件装的。

已经装入系统的字体

Academy Engraved LET	Chasm	Lucida Sans	Smudger Alts LET
Albertus	Clarendon	Lucida Sans Unicode	Smudger LET
Albertus Extra Bold	Clarendon Condensed	Lynda Cursive	Square721 BT
Albertus Medium	Clarendon Extended	Mangal	Staccato222 BT
Amaze	Comic Sans MS	Marigold	Surfer
Antique Olive	Coronet	Marlett	Sylfaen
Antique Olive Compact	Courier New	Mekanik LET	Symbol
Arial	CourierPS	Microsoft Sans Serif	SymbolPS
Arial Black	Dolphin	Milano LET	Tahoma
Arial Narrow	Dotum	MingLiU	Times
Bangle	DotumChe	Mirror	Times New Roman
Bart	Estrangelo Edessa	MisterEarl BT	Tiranti Solid LET
Basemic	Eurasia	Monotype Corsiva	Trebuchet MS
Basemic Symbol	Flat Brush	MS Gothic	Tunga
Basemic Times	Franklin Gothic Medium	MS Mincho	Univers
Batang	Galant	MS Outlook	Univers Condensed
BatangChe	Garamond	MS PGothic	University Roman Alts LET
Bimini	Gautami	MS PMincho	University Roman LET
Blackletter686 BT	Gaze	MS Reference Sans Serif	Verdana
Book Antiqua	Georgia	MS Reference Specialty	Victorian LET
Bookman Old Style	Gulim	MS UI Gothic	Webdings
Bookshelf Symbol 7	GulimChe	MT Extra	Westwood LET
Broadway BT	Gungsuh	MV Boli	Wingdings
cajcd fnta0	GungsuhChe	New Century Schoolbook	Wingdings 2
cajcd fnta1	Helvetica	Nina	Wingdings 3
cajcd fnta4	Helvetica Narrow	Notes	ZWAAdobeF
cajcd fnta6	Highlight LET	Odessa LET	仿宋_GB2312
cajcd fnta7	HolidayPi BT	OldDreadfulNo7 BT	华文中宋
cajcd fnta8	Impact	One Stroke Script LET	华文仿宋
cajcd fntaa	ITC Avant Garde Gothic	Orange LET	华文彩云
cajcd fntac	ITC Avant Garde Gothic Demi	Palatino	华文新魏
cajcd fntae	ITC Bookman Demi	Palatino Linotype	华文细黑
cajcd fntbd	ITC Bookman Light	Paris	华文行楷
cajcd fntbz	ITC Zapf Chancery	ParkAvenue BT	宋体
cajcd fntdg	ITC Zapf Dingbats	PMingLiU	宋体-方正超大字符集
cajcd fnthx	John Handy LET	Pump Demi Bold LET	幼圆
cajcd fntlt	Jokerman Alts LET	Quixley LET	新宋体
cajcd fntra	Jokerman LET	Raavi	方正姚体
cajcd fntst	Kelt	Rage Italic LET	方正舒体
cajcd fnttab	Kingsoft Phonetic Plain	Ruach LET	楷体_GB2312
Calligraph421 BT	La Bamba LET	Scruff LET	隶书
Cataneo BT	Latha	Short Hand	黑体
Century Gothic	Letter Gothic	Shruti	
CG Omega	Liberate	Signs	
CG Times	Lucida Console	Simpson	

注：如果想用程序将这些名称写入一个文本文件，需要注意 ofstream 不支持宽字符串的流输出，可以用实例模板

```
typedef basic_ofstream<wchar_t, char_traits<wchar_t>> ofwstream;
来定义一个新的文件输出流类型。
```

因为 VC05 没有提供将中文字字符的 Unicode 编码转换为 GB 码的函数（经我试验，wcstombs[_s]不支持中文串的转换），所以才需采用宽字符串流。

即使这样，微软的文件流库（类和函数）还是不支持中文字字符串的流输出。解决办法之一是，采用 CFile，但要注意宽字符串采用的是 UTF-16 编码，需要在文本文件的开始处，添加 0xFE 和 0xFF 这两个字节表示的编码标志。

(2) 字体 Font

字体类 Font 的构造函数有 6 个：

```
Font(const FontFamily *family, REAL emSize, INT style = FontStyleRegular, Unit unit
      = UnitPoint);
Font(const WCHAR *familyName, REAL emSize, INT style = FontStyleRegular, Unit
      unit = UnitPoint, const FontCollection *fontCollection = NULL);
Font(HDC hdc, const HFONT hfont);
Font(HDC hdc, const LOGFONTA *logfont);
Font(HDC hdc, const LOGFONTW *logfont);
Font(HDC hdc);
```

最常用的是前面两个构造函数。其中的第一个构造函数，其第一个输入参数是字体族的指针，所以必须先创建字体族对象。而第 2 个构造函数的第一个输入参数则是字体族(字样)的名称，不需要创建字体族对象，并且还多了可以选择的字体集作为最后一个输入参数。其余的构造函数都与 API 中的字体句柄、逻辑结构和 DC 中的当前字体有关，这些我们在 GDI 中已经讨论过。

下面我们重点讨论第二个构造函数的使用：

```
Font(const WCHAR *familyName, REAL emSize, INT style = FontStyleRegular, Unit
      unit = UnitPoint, const FontCollection *fontCollection = NULL);
```

其中：

(3) 字体种类

- familyName (字体族名) —— 宽字符串表示的字体名称

◆ 常用的英文字体族名有：

- Times New Roman: **Font Family Name** 字体族名 (有衬线)
- Arial: **Font Family Name** 字体族名 (无衬线)
- Arial Narrow: **Font Family Name** 字体族名 (窄体)
- Courier New: **Font Family Name** 字体族名 (等宽)

◆ 常用的中文字体族名有：

- 宋体: Font Family Name 字体族名 (正文)
 - 楷体_GB2312: Font Family Name 字体族名 (正文、标题)
 - 黑体: **Font Family Name** 字体族名 (标题、美术)
 - 仿宋_GB2312: Font Family Name 字体族名 (标题、美术)
- ◆ 其他中文字体有:
- 华文中宋: **Font Family Name** 字体族名
 - 华文仿宋: Font Family Name 字体族名
 - 华文彩云: **Font Family Name** 字体族名
 - 华文新魏: **Font Family Name** 字体族名
 - 华文细黑: **Font Family Name** 字体族名
 - 华文行楷: **Font Family Name** 字体族名
- 宋体-方正超大字符集: **Font Family Name** 字体族名
- 幼圆: Font Family Name 字体族名
 - 新宋体: Font Family Name 字体族名
 - 方正姚体: **Font Family Name** 字体族名
 - 方正舒体: **Font Family Name** 字体族名
 - 隶书: **Font Family Name** 字体族名
- ◆ 比较有意思的英文字体有:
- Academy Engraved LET: **Font Family Name**
 - Amaze: **Font Family Name**
 - Arial Black: **Font Family Name**
 - Bart: **Font Family Name**

- Blackletter686 BT: *Font Family Name*
- Chasm: Font Family Name
- Dolphin: Font Family Name
- Flat Brush: Font Family Name
- Gaze: Font Family Name
- Highlight LET: Font Family Name
- John Handy LET: Font Family Name
- Jokerman LET: Font Family Name
- Kelt: Font Family Name
- Liberate: Font Family Name
- Lynda Cursive: Font Family Name
- Marigold: *Font Family Name*
- Milano LET: Font Family Name
- MisterEarl BT: *Font Family Name*
- OldDreadfulNo7 BT: Font Family Name
- One Stroke Script LET: Font Family Name
- Orange LET: Font Family Name
- Paris: Font Family Name
- ParkAvenue BT: *Font Family Name*
- Pump Demi Bold LET: Font Family Name
- Quixley LET: Font Family Name
- Rage Italic LET: Font Family Name

- Ruach LET: **Font Family Name**
- Scruff LET: **Font Family Name**
- Short Hand: **Font Family Name**
- Simpson: **Font Family Name**
- Smudger LET: **Font Family Name**
- Staccato222 BT: ***Font Family Name***
- Surfer: **Font Family Name**
- Tiranti Solid LET: **Font Family Name**
- University Roman Alts LET: **Font Family Name**
- University Roman LET: **Font Family Name**
- Victorian LET: **Font Family Name**
- Westwood LET: **Font Family Name**

(4) 字体风格 FontStyle

- style (风格) —— 字体的风格, 可以取如下枚举常量 (缺省为 FontStyleRegular):

```
typedef enum {
    FontStyleRegular = 0, // 正常 (缺省值)
    FontStyleBold = 1, // 粗体
    FontStyleItalic = 2, // 斜体
    FontStyleBoldItalic = 3, // 粗斜体
    FontStyleUnderline = 4, // 下划线
    FontStyleStrikeout = 8 // 删除线
} FontStyle;
```

(5) 字体单位 Unit 与大小

- emSize (大小) 与 unit (单位) 有关, 可用单位有:

```
typedef enum {
    UnitWorld = 0, // 逻辑单位 (非物理单位, 缺省为像素)
    UnitDisplay = 1, // 设备单位, 如对显示器为像素、对打印机为墨点
    UnitPixel = 2, // 像素 (1/54 或 1/96 英寸? 与屏幕大小和分辨率有关)
}
```

```

    UnitPoint = 3, // 点或 1/72 英寸（缺省值）
    UnitInch = 4, // 英寸
    UnitDocument = 5, // 1/300 英寸
    UnitMillimeter = 6 // 毫米 mm
} Unit;

```

其中，em=M，在印刷行业中表示一个西文印刷符号的全长或全宽。

在 GDI 的 CFont 部分，已经介绍了中文字号与英文磅数（相当于这里的 UnitPoint 点值）的关系，下面是中文字号与几种主要 Unit 单位的关系表：（设 1 像素=1/54 英寸）

中文字号与 Unit 单位的关系

汉字 字号	Pixel 像素	Point 点	Inch 英寸	Document 文档	Millimeter 毫米
特号	133.3333	100	1.3889	416.6667	35.2778
小特	80	60	0.8333	250	21.1667
初号	56	42	0.5833	175	14.8167
小初	48	36	0.5	150	12.7
一号	34.6667	26	0.3611	108.3333	9.1722
小一	32	24	0.3333	100	8.4667
二号	29.3333	22	0.3056	91.6667	7.7611
小二	24	18	0.25	75	6.35
三号	21.3333	16	0.2222	66.6667	5.6444
小三	20	15	0.2083	62.5	5.2917
四号	18.6667	14	0.1944	58.3333	4.9389
小四	16	12	0.1667	50	4.2333
五号	14	10.5	0.1458	43.75	3.7042
小五	12	9	0.125	37.5	3.175
六号	10	7.5	0.1042	31.25	2.6458
小六	8.6667	6.5	0.0903	27.0833	2.2931
七号	7.3333	5.5	0.0764	22.9167	1.9403
八号	6.6667	5	0.0694	20.8333	1.7639

例如：

```

REAL fs[] = { 100, 60, 42, 36, 26, 24, 22, 18, 16, 15, 14, 12, 10.5, 9, 7.5, 6.5, 5.5, 5 };
CString fno[] = { L"特", L"小特", L"初", L"小初", L"一", L"小一", L"二", L"小二",
    L"三", L"小三", L"四", L"小四", L"五", L"小五", L"六", L"小六", L"七", L"八" };
wchar_t str[100];
REAL size, y = 10.0f;
SolidBrush textBrush(Color::Black);
Graphics graph(pDC->m_hDC);
for (int i = 0; i < 18; i++) {
    size = fs[i];
    swprintf_s(str, 100, L"这是%s 号字(% .4g 像素 %g 点 % .4g 英寸 % .4g 文档 % .4g 毫米)",
        fno[i], size * 4 / 3.0, size, size / 72.0, size * 300 / 72.0, size / 72.0 * 25.4);
}

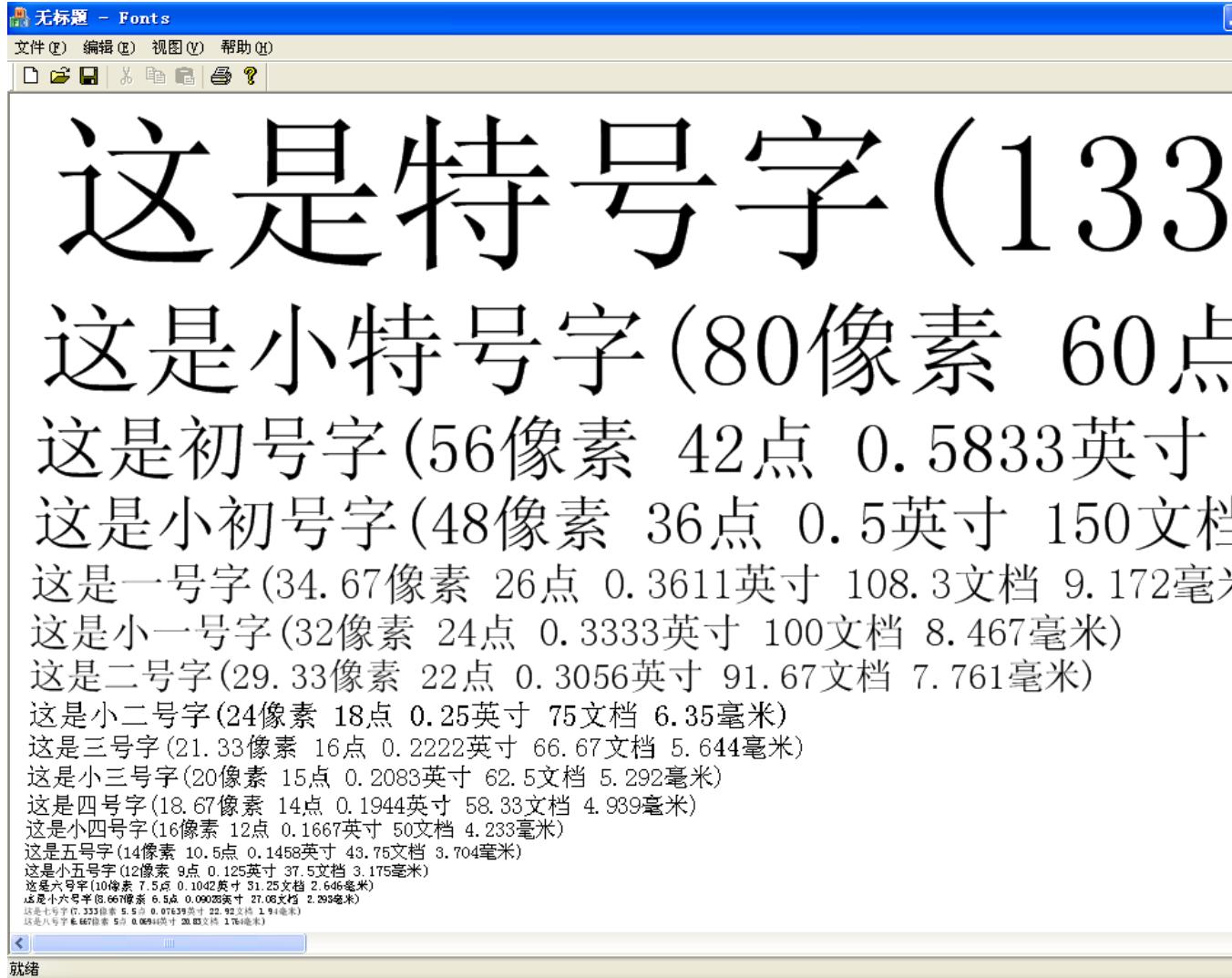
```

```

graph.DrawString(str, INT(wcslen(str)), &Font(L"宋体", size), PointF(10.0f, y), &textBrush);
y += size * 1.5f;
}

```

输出结果如：



2) 绘制文本

在 GDI 中，我们用 CDC 类的成员函数 TextOut、DrawText 和 ExtTextOut 等来输出文本串。在 GDI+中，我们则是利用 Graphics 类的重载成员函数 DrawString 来绘制文本。

(1) 画串函数 DrawString

```

Status DrawString(const WCHAR *string, INT length, const Font *font, const PointF
&origin, const Brush *brush);
Status DrawString(const WCHAR *string, INT length, const Font *font, const PointF
&origin, const StringFormat *stringFormat, const Brush *brush);

```

```
Status DrawString(const WCHAR *string, INT length, const Font *font, const RectF  
&layoutRect, const StringFormat *stringFormat, const Brush *brush);
```

这三个同名的 Graphics 类重载成员函数，都以宽字符串作为第一个输入参数（不支持普通字符串）、串长为第二个参数（对以 null 结尾的字符串，可以使用-1 来代替）、最后一个参数则都是绘制文本用的画刷指针。

不同的是第三个输入参数（都是浮点数版本，不支持整数版本）：前两个函数的是浮点数版的点类 PointF 对象，表示文本串的位置（缺省是左上角）；最后一个函数的是浮点数版的矩形类 RectF 对象，表示绘制文本的范围（超出部分会被截掉）。

另一个不同之处是，后两个函数比第一个函数多了一个输入参数——串格式类 StringFormat 对象的指针，用于设置文本的对齐方式、输出方向、自动换行、制表符定制、剪裁等。

第一个画串函数最简单，使用得也最多。例如：

```
graph.DrawString(str, INT(wcslen(str)), &font(L"宋体", 12), PointF(10.0f, 10.0f),  
&brush);  
graph.DrawString(str, -1, &font, &rect, &stringFormat, &brush);
```

(2) 串格式类 StringFormat

StringFormat 是从 GdiplusBase 类派生的一个 GDI+类，用于设置绘制字符串时的各种格式。其主要的构造函数为：

```
StringFormat(INT formatFlags = 0, LANGID language = LANG_NEUTRAL);
```

其中：

- formatFlags（格式标志位）—— 用于设置各种输出格式，取值为 StringFormatFlags 枚举的下列常量之位或：

```
typedef enum {  
    StringFormatFlagsDirectionRightToLeft = 0x00000001, // 方向从右到左（缺省为从左到右）  
    StringFormatFlagsDirectionVertical = 0x00000002, // 垂直方向（缺省为水平）  
    StringFormatFlagsNoFitBlackBox = 0x00000004, // 允许字符尾部悬于矩形之外  
    StringFormatFlagsDisplayFormatControl = 0x00000020, // Unicode 布局控制符起作用  
    StringFormatFlagsNoFontFallback = 0x00000400, // 有替换用的“缺少字体”（缺省为开方形符）  
    StringFormatFlagsMeasureTrailingSpaces = 0x00000800, // 测量时包含尾部空格符（缺省不包含）  
    StringFormatFlagsNoWrap = 0x00001000, // 不自动换行  
    StringFormatFlagsLineLimit = 0x00002000, // 最后一行必须为整行高，避免半行高的输出  
    StringFormatFlagsNoClip = 0x00004000 // 不使用剪裁  
} StringFormatFlags;
```

- language（语言）—— 取值为 16 位语言标识符类型 LANGID，缺省值为 LANG_NEUTRAL（语言中立），表示采用用户的缺省语言。

(3) 语言 LANGID

可以使用 MAKELANGID 宏：

```
WORD MAKELANGID(  
    USHORT usPrimaryLanguage, // primary language identifier  
    USHORT usSubLanguage // sublanguage identifier
```

);

来设置语言标识符。其中常用的预定义用户主语言有：

Identifier	Predefined symbol	Language
0x00	LANG_NEUTRAL	Neutral 中立
0x01	LANG_ARABIC	Arabic 阿拉伯语
0x04	LANG_CHINESE	Chinese 汉语
0x07	LANG_GERMAN	German 德语
0x09	LANG_ENGLISH	English 英语
0x0a	LANG_SPANISH	Spanish 西班牙语
0x0c	LANG_FRENCH	French 法语
0x10	LANG_ITALIAN	Italian 意大利语
0x11	LANG_JAPANESE	Japanese 日语
0x12	LANG_KOREAN	Korean 朝鲜语
0x19	LANG_RUSSIAN	Russian 俄语

常用的预定义用户子语言有：

Identifier	Predefined symbol	Language
0x00	SUBLANG_NEUTRAL	Language neutral 语言中立
0x01	SUBLANG_DEFAULT	User Default 用户缺省
0x02	SUBLANG_SYS_DEFAULT	System Default 系统缺省
0x01	SUBLANG_CHINESE_TRADITIONAL	Chinese (Traditional)繁体中文
0x02	SUBLANG_CHINESE_SIMPLIFIED	Chinese (Simplified)简体中文
0x03	SUBLANG_CHINESE_HONGKONG	Chinese (Hong Kong SAR, PRC)香港中文
0x04	SUBLANG_CHINESE_SINGAPORE	Chinese (Singapore)新加坡中文
0x05	SUBLANG_CHINESE_MACAU	Chinese (Macao SAR, PRC)澳门中文
0x01	SUBLANG_ENGLISH_US	English (US)美国英语
0x02	SUBLANG_ENGLISH_UK	English (UK)英国英语
0x03	SUBLANG_ENGLISH_AUS	English (Australian)澳大利亚英语
0x04	SUBLANG_ENGLISH_CAN	English (Canadian)加拿大英语

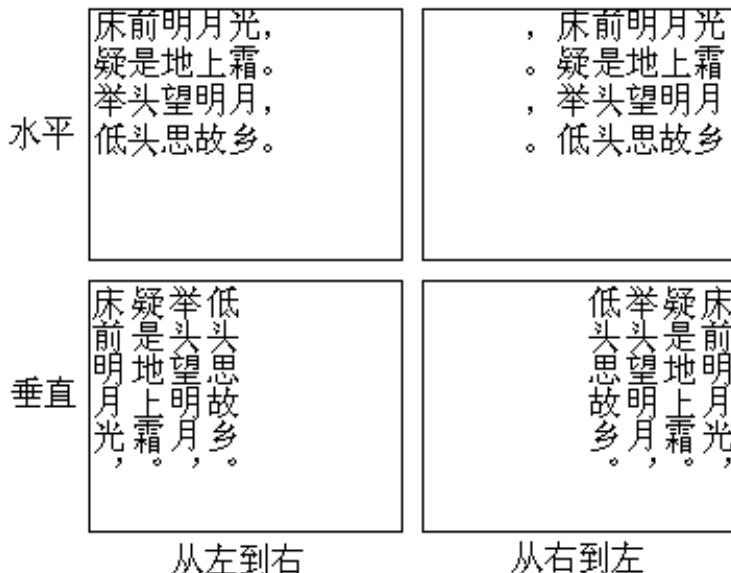
例如：（经我试验，对显示输出，无改变效果）

(4) 输出方向

缺省的文本串输出方向是从左到右水平绘制。也可以在 `StringFormat` 类的构造函数中使用参数值: `StringFormatFlagsDirectionRightToLeft` 和 `StringFormatFlagsDirectionVertical` 来修改文本串的输出方向为从右到左水平绘制和从上到下垂直绘制。例如:

```
Graphics graph(pDC->m_hDC);
Pen pen(Color::Black);
SolidBrush textBrush(Color::Black);
REAL w = 150.0f, h = 120.0f, d = 10.0f;
REAL x1 = d, x2 = x1 + w + d;
REAL y1 = d, y2 = y1 + h + d;
SizeF size(w, h);
RectF rect1(PointF(x1, y1), size), rect2(PointF(x2, y1), size),
rect3(PointF(x1, y2), size), rect4(PointF(x2, y2), size);
Font font(L"宋体", 12);
CString str = L"床前明月光，\r\n疑是地上霜。 \r\n举头望明月，\r\n低头思故乡。";
graph.DrawRectangle(&pen, rect1);
StringFormat stringFormat1; // 缺省为从左到右水平输出
graph.DrawString(str, str.GetLength(), &font, rect1, &stringFormat1, &textBrush);
graph.DrawRectangle(&pen, rect2);
StringFormat stringFormat2(StringFormatFlagsDirectionRightToLeft);
graph.DrawString(str, str.GetLength(), &font, rect2, &stringFormat2, &textBrush);
graph.DrawRectangle(&pen, rect3);
StringFormat stringFormat3(StringFormatFlagsDirectionVertical);
graph.DrawString(str, str.GetLength(), &font, rect3, &stringFormat3, &textBrush);
graph.DrawRectangle(&pen, rect4);
StringFormat stringFormat4(StringFormatFlagsDirectionRightToLeft |
StringFormatFlagsDirectionVertical);
graph.DrawString(str, str.GetLength(), &font, rect4, &stringFormat4, &textBrush);
```

输出结果为:



(5) 剪裁与换行

缺省情况下，使用矩形输出长文本串时，会自动换行和剪裁。但是也可以在 StringFormatF 类的构造函数中，利用第一个输入参数的 StringFormatFlags 枚举值，来改变默认的设置。

```
Graphics graph(pDC->m_hDC);
Pen pen(Color::Black);
SolidBrush textBrush(Color::Black);
StringFormat stringFormat;
stringFormat.SetAlignment(StringAlignmentCenter);
Font font0(L"宋体", 12), font(L"宋体", 18);
CString str = L"这是一个长字符串，用于串格式构造函数中格式标志位参数的使用，可以设置剪裁和换行等。";
REAL w = 280.0f, h = 100.0f, d = 10.0f, ds = 40;
REAL x1 = d, x2 = x1 + w + d, y1 = d, y2 = y1 + h + d + ds;
SizeF size(w, h);
RectF rect1(PointF(x1, y1), size), rect2(PointF(x2, y1), size),
rect3(PointF(x1, y2), size), rect4(PointF(x2, y2), size);
graph.DrawRectangle(&pen, rect1);
StringFormat stringFormat1; // 缺省串格式
graph.DrawString(str, str.GetLength(), &font, rect1, &stringFormat1, &textBrush);
graph.DrawString(L"缺省值 0", -1, &font0, PointF(x1 + w / 2, y1 + h + 10),
&stringFormat, &textBrush);
graph.DrawRectangle(&pen, rect2);
StringFormat stringFormat2(StringFormatFlagsNoWrap);
graph.DrawString(str, str.GetLength(), &font, rect2, &stringFormat2, &textBrush);
graph.DrawString(L"不换行 StringFormatFlagsNoWrap", -1, &font0,
PointF(x2 + w / 2, y1 + h + 10), &stringFormat, &textBrush);
graph.DrawRectangle(&pen, rect3);
StringFormat stringFormat3(StringFormatFlagsLineLimit);
graph.DrawString(str, str.GetLength(), &font, rect3, &stringFormat3, &textBrush);
graph.DrawString(L"行限制 StringFormatFlagsLineLimit", -1, &font0,
PointF(x1 + w / 2, y2 + h + 10), &stringFormat, &textBrush);
graph.DrawRectangle(&pen, rect4);
StringFormat stringFormat4(StringFormatFlagsNoClip);
graph.DrawString(str, str.GetLength(), &font, rect4, &stringFormat4, &textBrush);
graph.DrawString(L"不剪裁 StringFormatFlagsNoClip", -1, &font0,
PointF(x2 + w / 2, y2 + h + 10), &stringFormat, &textBrush);
```

输出结果为：

这是一个长字符串，用于串格式构造函数中格式标志位参数的使用，可以设置剪裁和换行等。

缺省值0

这是一个长字符串，用于

不换行 StringFormatFlagsNoWrap

这是一个长字符串，用于串格式构造函数中格式标志位参数的使用，可以设置剪裁和换行等。

行限制 StringFormatFlagsLineLimit

这是一个长字符串，用于串格式构造函数中格式标志位参数的使用，可以设置剪裁和换行等。

不剪裁 StringFormatFlagsNoClip

(6) 对齐

可以通过 StringFormat 类的成员函数来设置输出文本串的对齐方式：

```
Status SetAlignment(StringAlignment align); // 设置水平对齐  
Status SetLineAlignment(StringAlignment align); // 设置垂直对齐
```

其中的输入参数为枚举常量：

```
typedef enum {  
    StringAlignmentNear = 0, // 靠近（左上）  
    StringAlignmentCenter = 1, // 中心（对中）  
    StringAlignmentFar = 2 // 远离（右下）  
} StringAlignment;
```

例如：

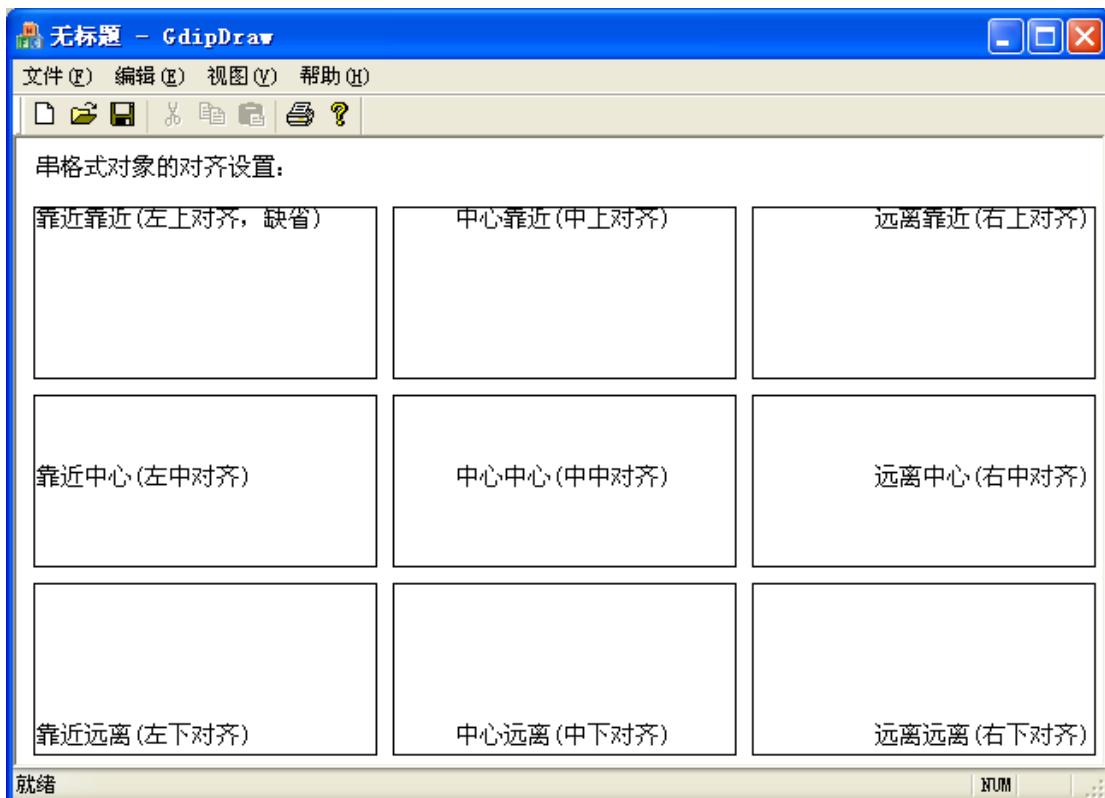
```
Graphics graph(pDC->m_hDC);  
Pen pen(Color::Black);  
SolidBrush textBrush(Color::Black);  
Font font(L"宋体", 10.5);  
CString str = L"串格式对象的对齐设置：";  
graph.DrawString(str, str.GetLength(), &font, PointF(10.0f, 10.0f), &textBrush);  
StringFormat stringFormat;  
StringAlignment hsa, vsa;  
CString astrs[3] = {L"靠近", L"中心", L"远离"};  
CString hstrs[3] = {L"左", L"中", L"右"};  
CString vstrs[3] = {L"上", L"中", L"下"};  
REAL w = 200.0f, h = 100.0f, d = 10.0f;  
SizeF size(w, h);  
REAL x, y;  
for (int i = 0; i < 9; i++) {  
    x = d + (w + d) * (i % 3);  
    y = 30.0f + d + (h + d) * (i / 3);  
}
```

```

    RectF rect(PointF(x, y), size);
    graph.DrawRectangle(&pen, rect);
    hsa = StringAlignment(i % 3);
    vsa = StringAlignment(i / 3);
    stringFormat.SetAlignment(hsa);
    stringFormat.SetLineAlignment(vsa);
    CString str = astrs[hsa] + astrs[vsa] + L "(" + hstrs[hsa] + vstrs[vsa] + L "对齐"
        + (i == 0 ? L ", 缺省" : L "") + L ")";
    graph.DrawString(str, str.GetLength(), &font, rect, &stringFormat, &textBrush);
}

```

输出结果为：



(7) 美术字

下面介绍几种，利用不同颜色、条纹和渐变的画刷以及多次绘图的方法，来达到一定美术效果的字符串绘制方法。

● 阴影字

可以使用两种不同颜色的画刷，经过在不同的位置多次绘制同一文本串，就可以达到输出阴影字的效果。例如：

```

Graphics graph(pDC->m_hDC);
SolidBrush textBrush(Color::Red), shadowBrush(Color::Gray);

```

```

HatchBrush hatchBrush(HatchStyleForwardDiagonal, Color::Black, Color::White);
CString str = L"阴影字符串";
Font font(L"华文新魏", 100);
REAL d = 10.0f, dd = 5.0f;
graph.DrawString(str, str.GetLength(), &font, PointF(d + dd, d + dd), &shadowBrush);
graph.DrawString(str, str.GetLength(), &font, PointF(d, d), &textBrush);
for (int i = 0; i < 20; i++)
    graph.DrawString(str, str.GetLength(), &font,
                    PointF(d + i, 150 + d + i + 2), &hatchBrush);
graph.DrawString(str, str.GetLength(), &font, PointF(d, 150 + d), &textBrush);

```

输出结果为：



● 条纹字

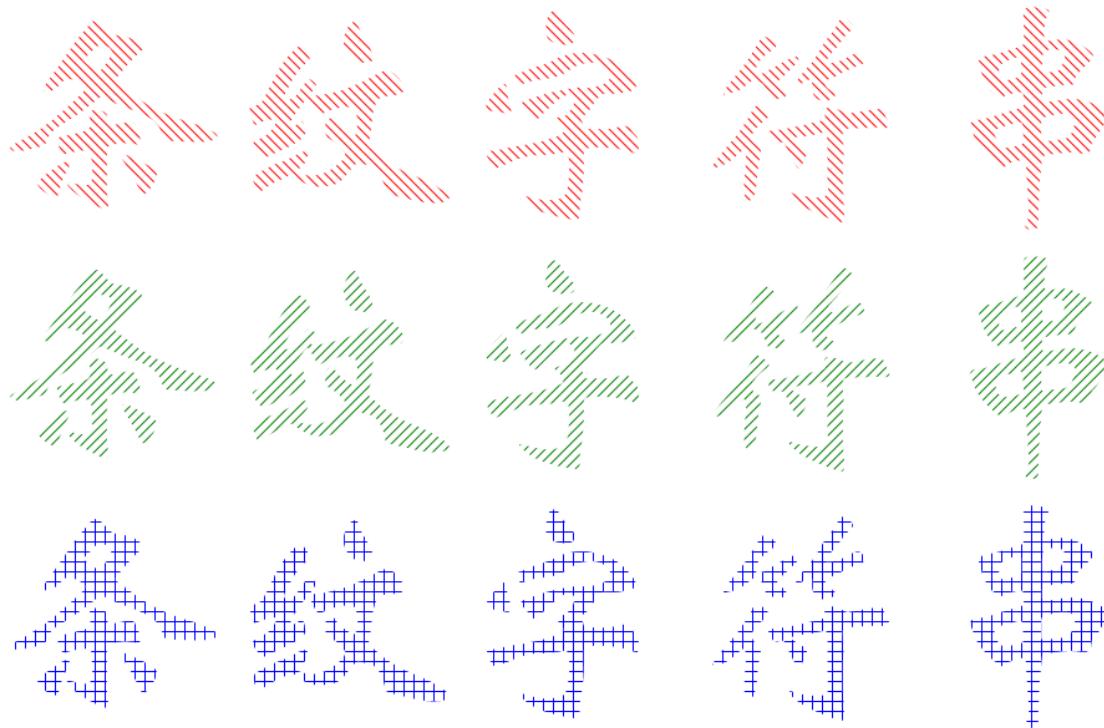
也可以直接利用条纹刷，来绘制条纹状的字符串。例如：

```

Graphics graph(pDC->m_hDC);
CString str = L"条纹字符串";
Font font(L"华文新魏", 140);
HatchBrush hatchBrush1(HatchStyleForwardDiagonal, Color::Red, Color::White);
graph.DrawString(str, str.GetLength(), &font, PointF(0.0f, 0.0f), &hatchBrush1);
HatchBrush hatchBrush2(HatchStyleBackwardDiagonal, Color::Green, Color::White);
graph.DrawString(str, str.GetLength(), &font, PointF(0.0f, 200.0f), &hatchBrush2);
HatchBrush hatchBrush3(HatchStyleCross, Color::Blue, Color::White);
graph.DrawString(str, str.GetLength(), &font, PointF(0.0f, 400.0f), &hatchBrush3);

```

输出结果为：



● 纹理字

还可以利用纹理刷来绘制纹理字符串。例如：

```
Graphics graph(pDC->m_hDC);
CString str = L"纹理字符串";
Font font(L"华文新魏", 140);
TextureBrush textureBrush(&Image(L"张东健.bmp"));
graph.DrawString(str, str.GetLength(), &font, PointF(10.0f, 10.0f), &textureBrush);
```

输出结果为：



● 渐变字

当然，也可以利用线性渐变刷来绘制色彩变幻的字符串。例如，使用前面的多色渐变代码，可以得到很好的变色效果：

```
Graphics graph(pDC->m_hDC);
CString str = L"颜色渐变字符串";
Font font(L"华文新魏", 100);
Color cols[] = {Color::Red, Color::Orange, Color::Yellow, Color::Green, Color::Cyan,
    Color::Blue, Color::Purple, Color::Magenta};
REAL bps[] = {0.0f, 0.15f, 0.3f, 0.45f, 0.6f, 0.75f, 0.875f, 1.0f};
LinearGradientBrush brush(Point(10, 10), Point(810, 10), Color::Black, Color::White);
```

```
brush.SetInterpolationColors(cols, bps, 8);
graph.DrawString(str, str.GetLength(), &font, PointF(10.0f, 10.0f), &brush);
```

输出结果为：



- 空心字与彩心字

还可以利用 GDI+的路径和路径渐变刷，来绘制空心和彩心字符串。例如：

```
Graphics graph(pDC->m_hDC);
FontFamily ff(L"隶书");
wchar_t str[] = L"测试字符串";
REAL emSize = 120; // UnitWorld
graph.DrawString(str, -1, &font(L"隶书", emSize, FontStyleRegular, UnitWorld),
    PointF(0, 0), &SolidBrush(Color::Green));
GraphicsPath path, *pOutlinePath;
path.AddString(str, -1, &ff, FontStyleRegular, emSize, Point(0, 100), NULL); // 847 个点
Pen pen(Color::Red);
graph.DrawPath(&pen, &path);
pOutlinePath = path.Clone();
pOutlinePath->Outline();
PathGradientBrush pgBrush(pOutlinePath);
int n = pOutlinePath->GetPointCount(); // 1023 个点
Color *cols = new Color[n];
for (int i = 0; i < n; i++) cols[i] = Color(rand() % 255, rand() % 255, rand() % 255);
pgBrush.SetCenterColor(Color(rand() % 255, rand() % 255, rand() % 255));
pgBrush.SetSurroundColors(cols, &n);
graph.TranslateTransform(0.0f, 100.0f);
graph.FillPath(&pgBrush, &path);
```

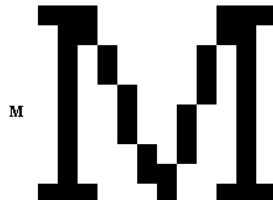
其中，由于用到了随机颜色，所以每次刷新时的颜色都不一样。输出结果为：



普通、空心和彩心字符串

(8) 平滑处理与 ClearType 技术

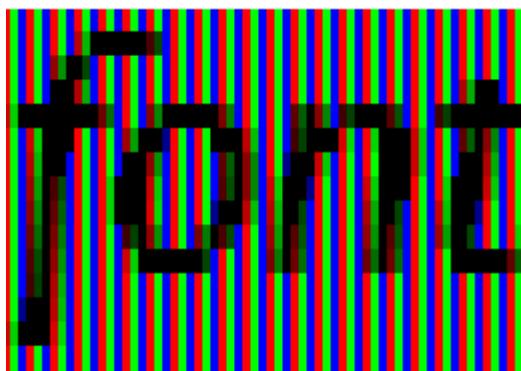
为了提高文字的清晰度，需要对绘制的文本串进行平滑处理，防止在（特别是点阵）文字被放大后出现明显的锯齿（马赛克 mosaic）现象。



ClearType（清晰活字）是微软公司于 1999 年 4 月 7 日推出的一种图形显示技术，主要用于改善 LCD（Liquid Crystal Display 液晶显示）显示器的显示效果，提高图形和文字的清晰度。

传统的 CRT 显示器，一般采用三只电子枪，将电子发射到荧光屏内壁的三色荧光粉上，利用光电三原色的加色原理产生出彩色图像。只要荧光粉有足够的密度，而且电子枪聚焦足够精确，图像就会很清晰。

LCD 虽然不存在电子枪聚焦问题，但是由于液晶显示器是由一个个单色液晶元件按红绿蓝列排列而成，而且液晶元件的大小是固定的，采用传统的采用灰色过渡的抗锯齿方法对 LCD 的效果不好。使用 ClearType 技术，采用彩色过渡的方法对字体轮廓进行平滑处理，抗锯齿效果有了明显地改善，可以大大提高 LCD 上的文字清晰度。



放大的 LCD 与 ClearType

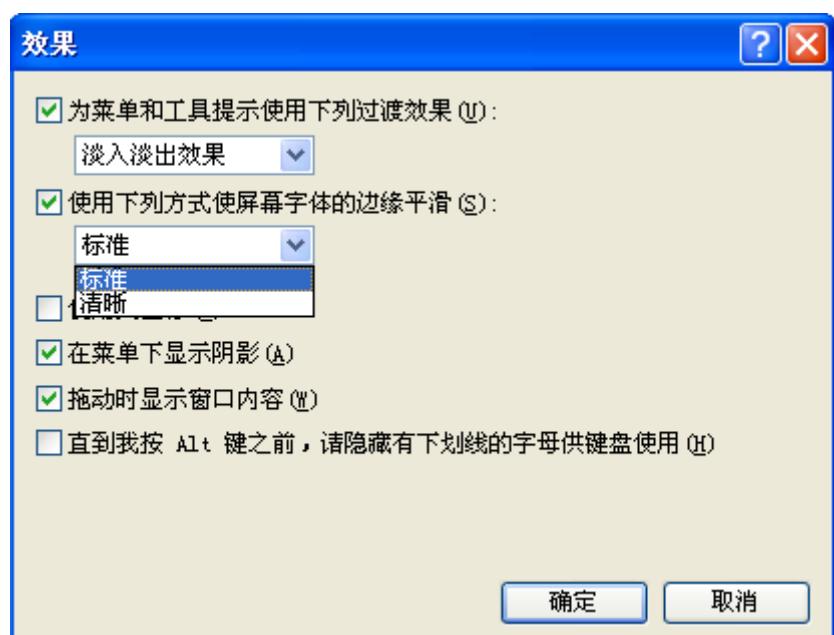


传统与 ClearType 两种抗锯齿方法的比较（放大 8 倍后的效果图）

该技术已经集成在 Windows XP 操作系统中。可以选中操作系统的“开始/设置/控制面板/显示”图标（或在桌面上按鼠标右键，在弹出菜单中选“属性”菜单项），打开“显示 属性”对话框（参见下图）：



选择“外观”页，单击“效果”按钮，弹出“效果”对话框：



在“使用下列方式使屏幕字体的边缘平滑”下拉列表中，选中“清晰”选项，按“确定”钮关闭“效果”对话框。再按“应用”钮和“确定”钮关闭“显示 属性”对话框。

下图是设置清晰前后的效果比较：



标准和清晰的区别（放大 8 倍后的效果图）

除了可以手工设置系统的抗锯齿处理方法外，也可以在 GDI+程序中，利用 Graphics 类的两个文本绘制提示成员函数：

```
TextRenderingHint GetTextRenderingHint(VOID) const;  
Status SetTextRenderingHint(TextRenderingHint newMode);
```

来获取和设置文字绘制时的平滑处理方法。其中的枚举类型 TextRenderingHint 的定义为：

```
typedef enum {  
    TextRenderingHintSystemDefault = 0, // 同系统平滑方式  
    TextRenderingHintSingleBitPerPixelGridFit = 1, // 不消锯齿，网格匹配  
    TextRenderingHintSingleBitPerPixel = 2, // 不消锯齿，不网格匹配  
    TextRenderingHintAntiAliasGridFit = 3, // 消锯齿，网格匹配  
    TextRenderingHintAntiAlias = 4, // 锯齿，不网格匹配  
    TextRenderingHintClearTypeGridFit = 5 // 使用 ClearType 技术，不网格匹配  
} TextRenderingHint;
```

其中的网格匹配 (grid fit)，主要是指在文本绘制时，通过调整字形的平直和垂直笔画的宽度，以达到提高文字输出质量的一种方法。

例如：

```
Graphics graph(pDC->m_hDC);  
SolidBrush textBrush(Color::Black);  
Font font(L"Arial", 16);  
CString str = L"font smoothing";  
wchar_t buf[5];  
for (int i = 0; i < 6; i++) {  
    _itow_s(i, buf, 5, 10);  
    graph.SetTextRenderingHint(TextRenderingHint(i));  
    graph.DrawString(buf, -1, &font, PointF(5.0f, 20.0f * i), &textBrush);  
    graph.DrawString(str, str.GetLength(), &font, PointF(30.0f, 20.0f * i),  
&textBrush);  
}
```

输出结果为：（放大 4 倍后的效果）

0 font smoothing
1 font smoothing
2 font smoothing
3 font smoothing
4 font smoothing
5 font smoothing

文字绘制时的平滑处理方式

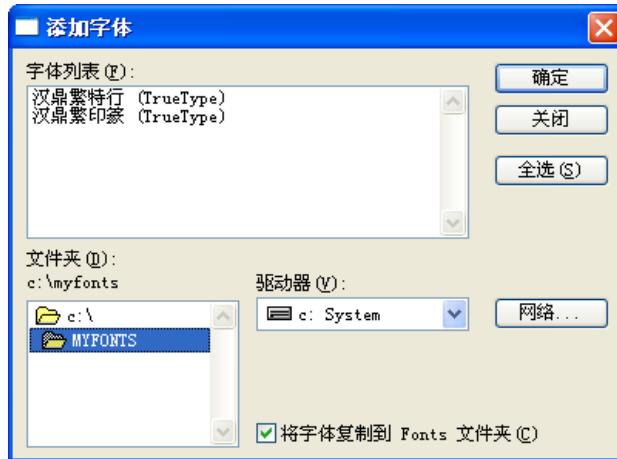
细心的同学可能会发现，除了渐变和路径刷外，大多数文本输出功能，GDI 都有。而且 GDI 还可以以任意角度绘制文本串，但是 GDI+好像不能。其实，这可以利用 GDI+的矩阵变换，很容易做到。矩阵变换的内容，会在第 6 小节介绍。

3) 专用字体集

如果你想使用系统中还没有被安装的字体，有如下两种方法可供选择。

(1) 手工安装字体

选择 Windows XP 操作系统的“开始/设置/控制面板/字体”图标，启动“字体”程序；然后再选择“文件/安装新字体”菜单项，打开“添加字体”对话框（参见下图）；选择字体所在的文件目录，会出现目录中所有字体的名称和类型列表；选中想安装的字体后，按确定关闭对话框。这样，就完成了字体的安装工作。将字体装进系统后，就可以和其他已装入字体一样正常使用了。



(2) 使用专用字体集

与已安装字体集类 `InstalledFontCollection` 一样，专用（私有）字体集类 `PrivateFontCollection` 也是字体集类 `FontCollection` 的派生类。

`PrivateFontCollection` 类只有一个缺省构造函数：

```
PrivateFontCollection(VOID);
```

构造一个空的字体集。但是它有两个专用成员函数：

```
Status AddFontFile(const WCHAR* filename);
```

```
Status AddMemoryFont(const VOID *memory, INT length);
```

可以用来往字体集中添加字体。常用的是第一个成员函数——添加字体文件。

将字体文件加入专用字体集后，我们就可以利用其父类的成员函数

```
Status GetLastStatus(VOID);
```

和 `GetFamilyCount`、`GetFamilies` 等，来判断装入是否成功、字体集中共有多少种字体、获取指定数目的字体族 `FontFamily` 对象的数组（指针）。这些都与在 1) “显示当前系统已装入的字体（族）名称”部分所讲的类似。包括用 `FontFamily` 类的 `GetFamilyName` 成员函数获取字体名称，并用该名称来创建字体对象（使用带字体集参数的构造函数），最后绘制文本串。

例如：（使用汉鼎繁印篆和汉鼎繁特行两种专用字体，输出文本串“专用字体集：字体名称”和诗句“三顾频烦天下计 两朝开济老臣心”。这两种字体所对应的字体文件 `HDZB_25.TTF` 和 `HDZB_16.TTF`，已经放入网络硬盘的资源子目录 `res` 中。你也可以自己从网上下载其他字体文件来进行试验。）

```
// 装入专用字体文件
PrivateFontCollection pfc;
pfc.AddFontFile(L"C:\\myFonts\\HDZB_16.TTF");
if(pfc.GetLastStatus() != Ok) {MessageBox(L"装入字体文件出错！"); return; }
pfc.AddFontFile(L"C:\\myFonts\\HDZB_25.TTF");
if(pfc.GetLastStatus() != Ok) {MessageBox(L"装入字体文件出错！"); return; }
int n = pfc.GetFamilyCount();
if (n < 2) {MessageBox(L"字体集中的字体数不够！"); return; }
// 获取字体族对象数组
FontFamily ffs[2];
```

```
int found;
pfc.GetFamilies(2, ffs, &found);
// 定义输出字符串
CString str0 = L"专用字体集: ", str;
CString str1 = L"三顾频烦天下计\r\n两朝开济老臣心";
// 设置中对齐
StringFormat stringFormat;
stringFormat.SetAlignment(StringAlignmentCenter);
RECT rect;
GetClientRect(&rect);
// 创建图形和文本刷对象
Graphics graph(pDC->m_hDC);
SolidBrush textBrush(Color::Black);
// 获取字体名称 1, 构造字体 1, 并输出字符串
wchar_t name[LF_FACESIZE];
ffs[0].GetFamilyName(name);
Font font1(name, 60, FontStyleRegular, UnitPixel, &pfc);
str = str0 + name;
graph.DrawString(str, str.GetLength(), &font1, PointF(0.0f, 0.0f), &textBrush);
graph.DrawString(str1, str1.GetLength(), &font1, PointF(rect.right / 2.0f, 80.0f),
    &stringFormat, &textBrush);
// 获取字体名称 2, 构造字体 2, 并输出字符串
ffs[1].GetFamilyName(name);
Font font2(name, 60, FontStyleRegular, UnitPixel, &pfc);
str = str0 + name;
graph.DrawString(str, str.GetLength(), &font2, PointF(0.0f, 220.0f), &textBrush);
graph.DrawString(str1, str1.GetLength(), &font2, PointF(rect.right / 2.0f, 300.0f),
    &stringFormat, &textBrush);
```

输出结果为：

專用字體集：漢鼎繁印篆

三顧頻煩天下計
兩朝開濟老臣心

專用字體集：漢鼎繁特行

三顧頻煩天下計
兩朝開濟老臣心

汉鼎繁印篆和汉鼎繁特行专用字体

4. 路径

路径 (path) 是一系列相互连接的直线和曲线，由许多不同类型的点所构成，用于表示复杂的不规则图形，也叫做图形路径 (graphics path)。路径可以被画轮廓和填充，也可以用于创建区域和路径渐变刷等。

在 GDI 中也有路径 (我们没有讲)，但是它只是作为 DC 的一种状态才能存在。独立的路径对象，则是 GDI+的新特点。

1) 图形路径 GraphicsPath

在 GDI+中，路径由图形路径类 GraphicsPath 表示，它也是图形基类 GraphicsBase 的派生类。

(1) 构造函数

GraphicsPath 类有三个构造函数：

```
GraphicsPath(FillMode fillMode = FillModeAlternate); // 构造一个空路径
GraphicsPath(const Point *points, const BYTE *types, INT count, FillMode fillMode =
FillModeAlternate); // 构造含指定整数型点数组的路径
GraphicsPath(const PointF *points, const BYTE *types, INT count, FillMode fillMode =
FillModeAlternate); // 构造含指定浮数型点数组的路径
```

其中：

- 填充模式参数 fillMode，我们在画填充多边形和曲线时已经讲过，枚举类型 FillMode

除了可取这里的缺省值“交替（Alternate）”之外，还有一个可取的值是“环绕（Winding）”：FillModeWinding；

- 点数组参数 points，可以是整数类型的，也可以是浮点数类型的；
- 点类型数组参数 types，主要点类型有路径起点、直线端点和贝塞尔点；
- 计数参数 count，为数组 points 和 types 中的元素数。这两个数组中的元素数必须一致。

(2) 点的种类

构造函数中，点的类型取值为枚举类型 PathPointType 常量；

```
typedef enum {
    PathPointTypeStart = 0, // 起点
    PathPointTypeLine = 1, // 直线端点
    PathPointTypeBezier = 3, // 贝塞尔（曲线的控制）点
    PathPointTypePathTypeMask = 0x7, // 点类型掩码（只保留低三位）
    PathPointTypePathDashMode = 0x10, // 未使用
    PathPointTypePathMarker = 0x20, // 标记点（用于路径分段）
    PathPointTypeCloseSubpath = 0x80, // 闭子路径（图形）的终点
    PathPointTypeBezier3 = 3 // 同 PathPointTypeBezier
} PathPointType;
```

其中，主要的点类型有起点、直线端点、贝塞尔点、标记点和闭子路径终点。其他曲线类型（如弧、椭圆和基数样条曲线等）在路径中都是用贝塞尔曲线来表示的。

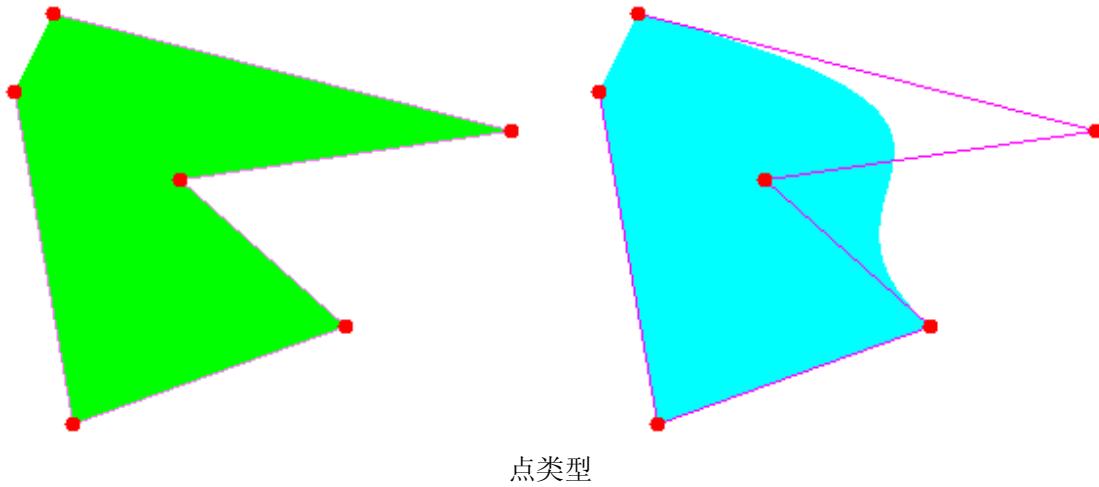
路径是由点组成的，但这里的点，不光指其坐标位置，还包括点的类型。同样的点坐标，不同的点类型，最后得到的路径可能大相径庭。

例如：（同一组点，定义两个路径，一个的点类型全是直线端点，另一个的起点之后有3个贝塞尔点，最后才是两个直线点）

```
Point points[] = {Point(40, 140), Point(275, 200), Point(105, 225),
    Point(190, 300), Point(50, 350), Point(20, 180)};
BYTE lineTypes[] = {PathPointTypeLine, PathPointTypeLine, PathPointTypeLine,
    PathPointTypeLine, PathPointTypeLine, PathPointTypeLine};
BYTE types[] = {PathPointTypeStart, PathPointTypeBezier, PathPointTypeBezier,
    PathPointTypeBezier, PathPointTypeLine, PathPointTypeLine};
GraphicsPath path1(points, lineTypes, 6), path2(points, types, 6);
Graphics graph(pDC->m_hDC);
graph.FillPath(&SolidBrush(Color::Lime), &path1);
graph.DrawLines(&Pen(Color::Violet), points, 6);
DrawPoints(graph, Color::Red, 4, points, 6);
graph.TranslateTransform(300.0f, 0.0f);
graph.FillPath(&SolidBrush(Color::Aqua), &path2);
graph.DrawLines(&Pen(Color::Magenta), points, 6);
DrawPoints(graph, Color::Red, 4, points, 6);
```

（其中的自定义画点列函数 DrawPoints，在画曲线时用过，源码参见 2.2）（9）小小节。）

输出结果为：



(3) 路径的构成

前面已经讲过，是路径是一系列相互连接的直线和曲线，它们最终都是由有序点列所组成。可以利用 **GraphicsPath** 类的后两个构造函数，将点数组直接加入路径中。但是，路径中的直线和曲线等图形，一般是通过调用路径类的若干添加图形成员函数给加进来的。

每个被加入的图形都可以是一个子路径（subpath）。路径对象，会将被加入图形（包括封闭图形）中的点尾首相接，连成一条完整的路径。

在路径中的图形都是开图形（起点和终点可能是同一个点，例如矩形、椭圆、多边形和闭曲线等），可以调用图形路径类的 **CloseFigure** 或 **CloseAllFigures** 成员函数：

```
Status CloseFigure(VOID); // 关闭当前子路径
Status CloseAllFigures(VOID); // 关闭所有子路径
```

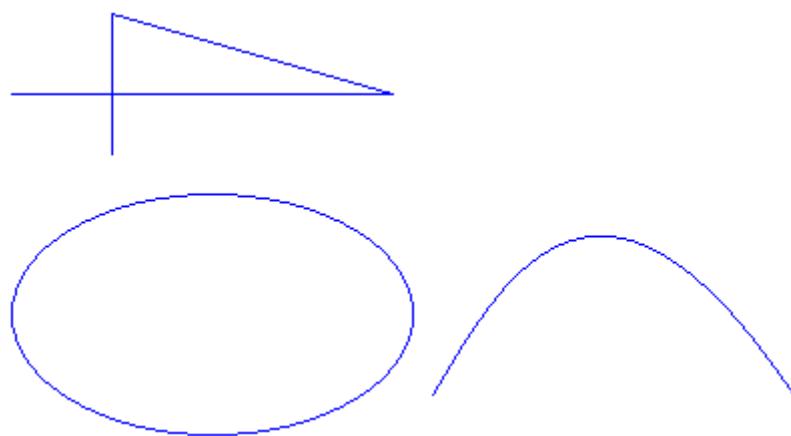
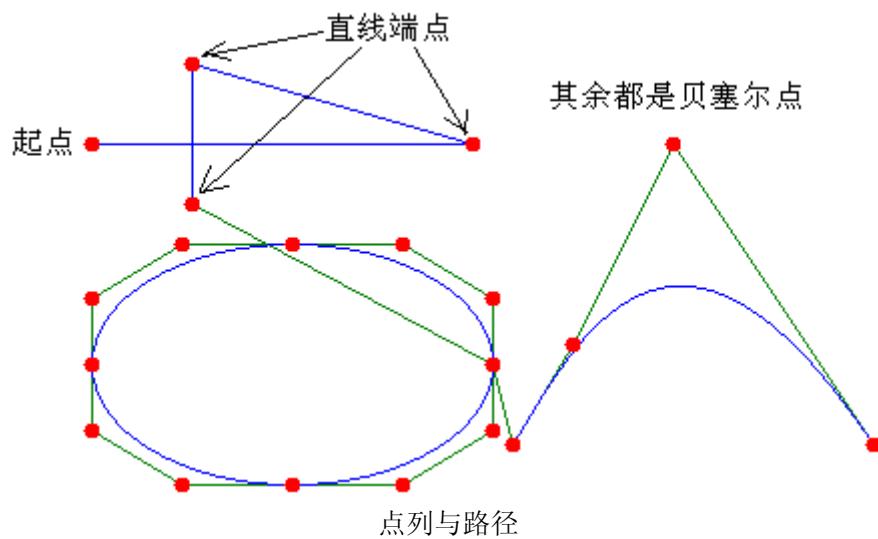
来显式闭合路径对象中的当前子路径或所有子路径。

例如：

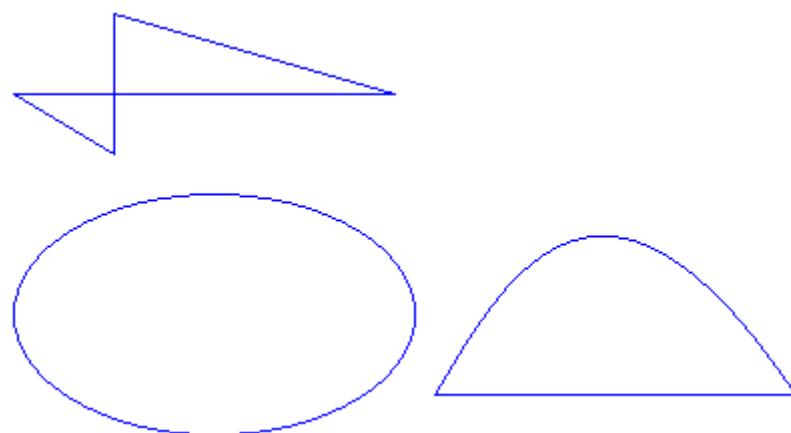
```
Graphics graph(pDC->m_hDC);
Pen pen(Color::Blue);
GraphicsPath path;
path.AddLine(10.0f, 50.0f, 200.0f, 50.0f); // 画平线
//path.StartFigure(); // 断开两条直线之间的连接（即分成两个子路径）
path.AddLine(60.0f, 10.0f, 60.0f, 80.0f); // 画垂线
path.AddEllipse(10, 100, 200, 120); // 画椭圆      //画贝塞尔曲线:
path.AddBezier(Point(220, 200), Point(250, 150), Point(300, 50), Point(400, 200));
int n = path.GetPointCount(); // 获取路径中的点数
Point *points = new Point[n];
path.GetPathPoints(points, n); // 获取路径中的点
//graph.FillPath(&SolidBrush(Color::Aqua), &path); // 填充（开）路径
graph.DrawLine(&Pen(Color::Green), points, n); // 画折线
//path.CloseAllFigures(); // 关闭所有子路径
graph.DrawPath(&pen, &path); // 画路径轮廓
DrawPoints(graph, Color::Red, 4, points, n); // 画路径中的点
```

（其中的自定义画点列函数 **DrawPoints**，在画曲线时用过，源码参见 2.2）（9）小小节。）

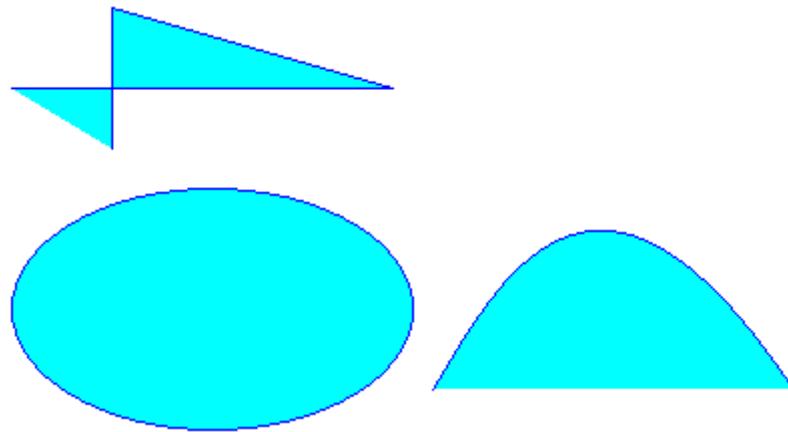
输出结果为：



开（子）路径



闭（子）路径



填充（开）路径

(4) 添加图形

图形路径类 GraphicsPath 中的下列成员函数，用于添加图形到路径中：

```
// 加直线
Status AddLine(INT x1, INT y1, INT x2, INT y2);
Status AddLine(const Point &pt1, const Point &pt2);
Status AddLine(REAL x1, REAL y1, REAL x2, REAL y2);
Status AddLine(const PointF &pt1, const PointF &pt2);

// 加折线
Status AddLines(const Point *points, INT count);
Status AddLines(const PointF *points, INT count);

// 加多边形
Status AddPolygon(const Point *points, INT count);
Status AddPolygon(const PointF *points, INT count);

// 加矩形
Status AddRectangle(const Rect &rect);
Status AddRectangle(const RectF &rect);

// 加矩形组
Status AddRectangles(const Rect *rects, INT count);
Status AddRectangles(const RectF *rects, INT count);

// 加弧
Status AddArc(INT x, INT y, INT width, INT height, REAL startAngle, REAL sweepAngle);
Status AddArc(const Rect &rect, REAL startAngle, REAL sweepAngle);
Status AddArc(REAL x, REAL y, REAL width, REAL height, REAL startAngle, REAL sweepAngle);
Status AddArc(const RectF &rect, REAL startAngle, REAL sweepAngle);

// 加饼
Status AddPie(INT x, INT y, INT width, INT height, REAL startAngle, REAL sweepAngle);
Status AddPie(const Rect &rect, REAL startAngle, REAL sweepAngle);
Status AddPie(REAL x, REAL y, REAL width, REAL height, REAL startAngle, REAL sweepAngle);
Status AddPie(const RectF &rect, REAL startAngle, REAL sweepAngle);

// 加椭圆
```

```

Status AddEllipse(INT x, INT y, INT width, INT height);
Status AddEllipse(const Rect &rect);
Status AddEllipse(REAL x, REAL y, REAL width, REAL height);
Status AddEllipse(const RectF &rect);
// 加贝塞尔曲线
Status AddBezier(INT x1, INT y1, INT x2, INT y2, INT x3, INT y3, INT x4, INT y4);
Status AddBezier(const Point &pt1, const Point &pt2, const Point &pt3, const Point &pt4);
Status AddBezier(REAL x1, REAL y1, REAL x2, REAL y2, REAL x3, REAL y3, REAL x4, REAL y4);
Status AddBezier(const PointF &pt1, const PointF &pt2, const PointF &pt3, const PointF &pt4);
// 加相连的多段贝塞尔曲线
Status AddBeziers(const Point *points, INT count);
Status AddBeziers(const PointF *points, INT count);
// 加基数样条曲线
Status AddCurve(const Point *points, INT count);
Status AddCurve(const Point *points, INT count, REAL tension);
Status AddCurve(const Point *points, INT count, INT offset, INT numberOfSegments, REAL tension);
Status AddCurve(const PointF *points, INT count);
Status AddCurve(const PointF *points, INT count, REAL tension);
Status AddCurve(const PointF *points, INT count, INT offset, INT numberOfSegments, REAL tension);
// 加闭基数样条曲线
Status AddClosedCurve(const Point *points, INT count);
Status AddClosedCurve(const Point *points, INT count, REAL tension);
Status AddClosedCurve(const PointF *points, INT count);
Status AddClosedCurve(const PointF *points, INT count, REAL tension);
// 加串
Status AddString(const WCHAR *string, INT length, const FontFamily *family, INT style, REAL
    emSize, const Point &origin, const StringFormat *format);
Status AddString(const WCHAR *string, INT length, const FontFamily *family, INT style, REAL
    emSize, const Rect &layoutRect, const StringFormat *format);
Status AddString(const WCHAR *string, INT length, const FontFamily *family, INT style, REAL
    emSize, const PointF &origin, const StringFormat *format);
Status AddString(const WCHAR *string, INT length, const FontFamily *family, INT style, REAL
    emSize, const RectF &layoutRect, const StringFormat *format);

```

(4) 子路径

每个被加入的闭图形（如矩形、椭圆、多边形、饼、闭基数样条曲线等），都是一个子路径（subpath）。而在缺省情况下，连续加入的所有开图形（如直线、折线、弧、贝塞尔曲线和基数样条曲线等），共同构成一个子路径。子路径也被叫做形状（figure）。

也可以利用 GraphicsPath 类的成员函数：

```

Status StartFigure(VOID); // 不关闭当前子路径，就开始新子路径
Status CloseFigure(VOID); // 先关闭当前子路径，再开始新子路径

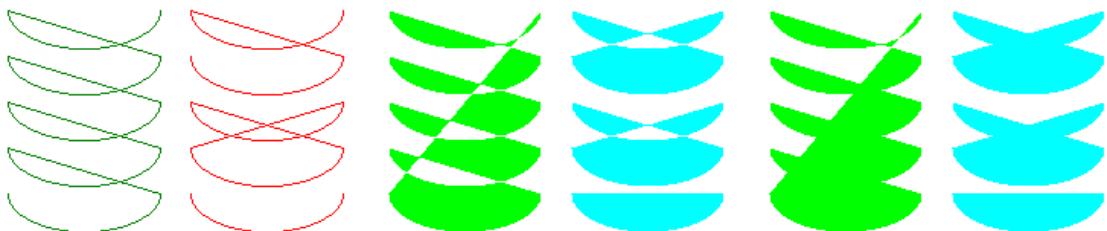
```

来开始一个新的子路径（即后面所添加的图形，属于另一个子路径）。

例如：

```
GraphicsPath path1, path2;
for (int i = 0; i < 5; i++) {
    path1.AddArc(10, i * 30, 100, 50, 0.0f, 180.0f);
    if (i == 2) path2.StartFigure();
    else if (i == 4) path2.CloseFigure();
    path2.AddArc(130, i * 30, 100, 50, 0.0f, 180.0f);
}
Graphics graph(pDC->m_hDC);
graph.DrawPath(&Pen(Color::Green), &path1);
graph.DrawPath(&Pen(Color::Red), &path2);
graph.TranslateTransform(250.0f, 0.0f);
graph.FillPath(&SolidBrush(Color::Lime), &path1);
graph.FillPath(&SolidBrush(Color::Aqua), &path2);
path1.SetFillMode(FillModeWinding);
path2.SetFillMode(FillModeWinding);
graph.TranslateTransform(250.0f, 0.0f);
graph.FillPath(&SolidBrush(Color::Lime), &path1);
graph.FillPath(&SolidBrush(Color::Aqua), &path2);
```

输出结果为：



(开/闭) 子路径与交替和环绕填充模式

可以利用 GDI+的图形路径迭代器类 GraphicsPathIterator 来获取路径中的子路径信息，并可遍历路径中的所有子路径。

该类只有一个构造函数，以路径指针为唯一的输入参数：

```
GraphicsPathIterator(GraphicsPath *path);
```

可以利用该类的成员函数

```
INT GetSubpathCount(VOID);
```

来获取路径中的子路径总数。

用其成员函数（我试过，不太听话）

```
INT NextPathType(BYTE *pathType, INT *startIndex, INT *endIndex);
```

获取下一个子路径中同类点的类型（直线或贝塞尔）和这些点在路径中的始末点位置，并返回子路径中的点数。

用 GraphicsPathIterator 类的成员函数

```
INT NextSubpath(GraphicsPath *path, BOOL *isClosed);
```

```
INT NextSubpath(INT *startIndex, INT *endIndex, BOOL *isClosed);
```

可以获取下一个子路径（的始末点位置），判断其是否关闭，并返回子路径中的点数。

在进行一次遍历操作后，需要调用下面的反绕成员函数，使迭代器返回路径的起点：

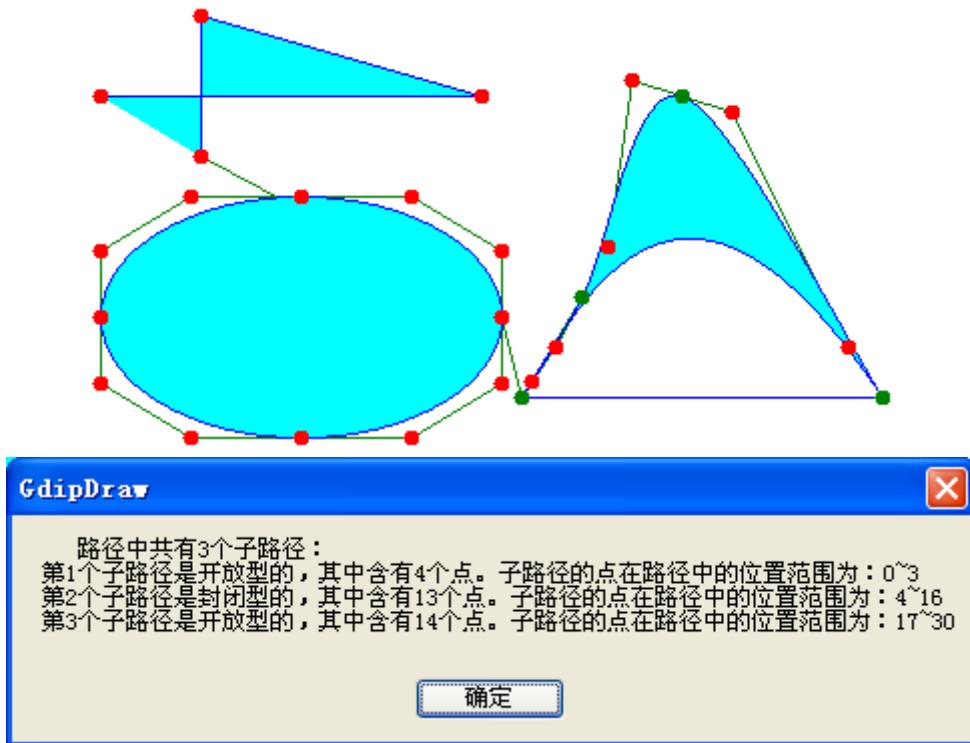
VOID Rewind(VOID);

例如：（由 4.1）（3）中的例子改造而成，增加了一条与贝塞尔曲线控制点相同的基数样条曲线。红色部分为修改过或新加的代码）

```
Graphics graph(pDC->m_hDC);
Pen pen(Color::Blue);
GraphicsPath path;
path.AddLine(10.0f, 50.0f, 200.0f, 50.0f);
path.AddLine(60.0f, 10.0f, 60.0f, 80.0f);
path.AddEllipse(10, 100, 200, 120);
Point ps[] = {Point(220, 200), Point(250, 150), Point(300, 50), Point(400, 200)};
path.AddBeziers(ps, 4);
path.AddCurve(ps, 4);
int n = path.GetPointCount();
Point *points = new Point[n];
path.GetPathPoints(points, n);
graph.DrawLines(&Pen(Color::Green), points, n);
//path.SetFillMode(FillModeWinding);
graph.FillPath(&SolidBrush(Color::Aqua), &path);
graph.DrawPath(&pen, &path);
DrawPoints(graph, Color::Red, 4, points, n);
DrawPoints(graph, Color::Green, 4, ps, 4);
GraphicsPathIterator pathItr(&path);
int m = pathItr.GetSubpathCount();
CString str;
wchar_t buf[100];
swprintf_s(buf, 100, L"    路径中共有%d 个子路径: \r\n", m);
str += buf;
INT sn, start, end;
BOOL isClosed;
for (int i = 0; i < m; i++) {
    sn = pathItr.NextSubpath(&start, &end, &isClosed);
    swprintf_s(buf, 100, L"第%d 个子路径是%ls 型的，其中含有%d 个点。 \
子路径的点在路径中的位置范围为： %d~%d\r\n",
               i + 1, isClosed ? L"封闭" : L"开放", sn, start, end);
    str += buf;
}
if (first) MessageBox(str);
```

其中的 first 为自定义的一个视图类的 bool 型类变量，初始化为 true，在首次调用 OnDraw 后再设为 false。用于避免系统反复调用 MessageBox 函数，造成死循环。（假设本程序段在 OnDraw 函数中被执行）

输出结果为：



可见，路径对象会把同一子路径中开图形的尾首相连。例如，开始两条直线所组成的第 1 个子路径，和最后两条曲线所组成的第 3 个子路径。

本来只有四个参数（左上角坐标和宽高）的椭圆，在路径中被变成了用 13 个点表示的贝塞尔曲线；本来只有四个点的基数样条曲线，在路径中也被变成了用 $(14 - 4 =)$ 10 个点表示的贝塞尔曲线。（其中 4 个绿色点为原始的控制点）

(5) 绘制路径

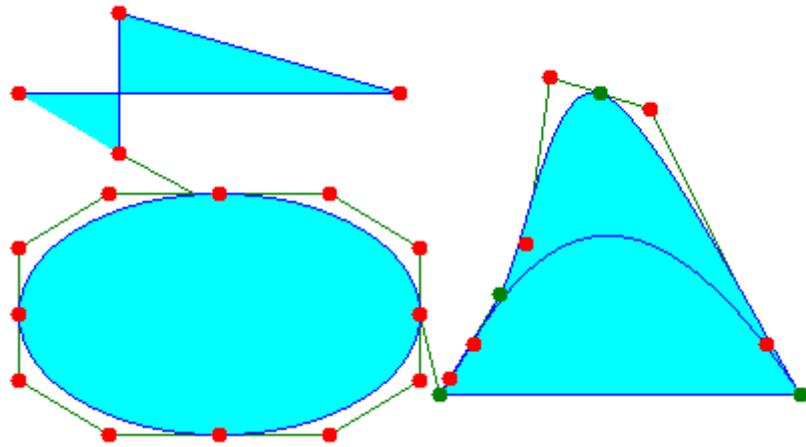
可以用 `Graphics` 类的成员函数

`Status DrawPath(const Pen *pen, const GraphicsPath *path);`
来画路径的轮廓。用其另一个成员函数

`Status FillPath(const Brush *brush, const GraphicsPath *path);`
来填充路径的内部（对开路径，会先自动封闭，然后再进行填充）。

当然你也可以用 `GraphicsPath` 类的成员函数：

`Status SetFillMode(FillMode fillmode);`
`FillMode GetFillMode(VOID);`
来设置不同的填充模式或者获取当前的填充模式。例如，如果去掉上例中
`//path.SetFillMode(FillModeWinding);`
前的注释符 “//”，则最后一个曲线子路径的填充效果就不一样了：



关于画路径轮廓和填充路径的例子，前面已经有了很多，这里就不再列举了。

(6) 获取点信息

在创建路径并添加各种几何图形或字符串之后，我们可以调用如下一些 `GraphicsPath` 类的成员函数，来获取路径中的点的信息。包括点的坐标信息和点的类型信息：

```
INT GetPointCount(VOID); // 获取路径中的总点数
Status GetPathPoints(Point *points, INT count); // 获取路径中（指定数目的）整数点数组
Status GetPathPoints(PointF *points, INT count); // 获取路径中（指定数目的）浮点数点数组
Status GetPathTypes(BYTE *types, INT count); // 获取路径中（指定数目的）点类型数组
```

例如：

```
.....
int n = path.GetPointCount();
Point *points = new Point[n];
path.GetPathPoints(points, n);
graph.DrawLine(&Pen(Color::Green), points, n);
graph.FillPath(&SolidBrush(Color::Aqua), &path);
graph.DrawPath(&pen, &path);
DrawPoints(graph, Color::Red, 4, points, n);
```

2) 路径渐变刷 PathGradientBrush

路径可以表示复杂的图形，可以用于绘制这些图形的轮廓和填充，也可以用于创建区域（在下一小节介绍）和颜色渐变刷。后者在前面美术字部分的彩心字符串例中（参见 3.2 (7)），我们已经用过。

与其它具体刷（如实心刷、条纹刷和纹理刷等）类一样，路径渐变（梯度）刷类 `PathGradientBrush`，也是 `Brush` 类的派生类。它有 3 个构造函数：

```
PathGradientBrush(const GraphicsPath *path);
PathGradientBrush(const Point *points, INT count, WrapMode wrapMode = WrapModeClamp);
PathGradientBrush(const PointF *points, INT count, WrapMode wrapMode = WrapModeClamp);
```

第一个构造函数从现有路径对象来创建画刷，后两个则是从整数或浮点数点集来直接创

建画刷，而且它们两个还有一个重复排列的输入参数 wrapMode，缺省值为 WrapModeClamp（不重复排列）。

路径刷的颜色，一般是从路径点（周边轮廓）向路径中心渐变。路径刷的缺省中心为路径的形心，可以用路径刷成员函数：

```
Status SetCenterPoint(const Point &point);
Status SetCenterPoint(const PointF &point);
```

来重新设置。其中的中心点，可以位于任何位置，包括在路径的范围之外。对应的获取刷中心的函数是：

```
Status GetCenterPoint(Point *point);
Status GetCenterPoint(PointF *point);
```

其它常用的路径刷成员函数有：

```
Status SetCenterColor(const Color &color); // 设置刷中心颜色
Status SetSurroundColors(const Color *colors, INT *count); // 设置路径点颜色
Status GetCenterColor(Color *color); // 获取刷中心颜色
INT GetSurroundColorCount(VOID); // 获取路径点颜色数目
Status GetSurroundColors(Color *colors, INT *count); // 获取路径点颜色数组
```

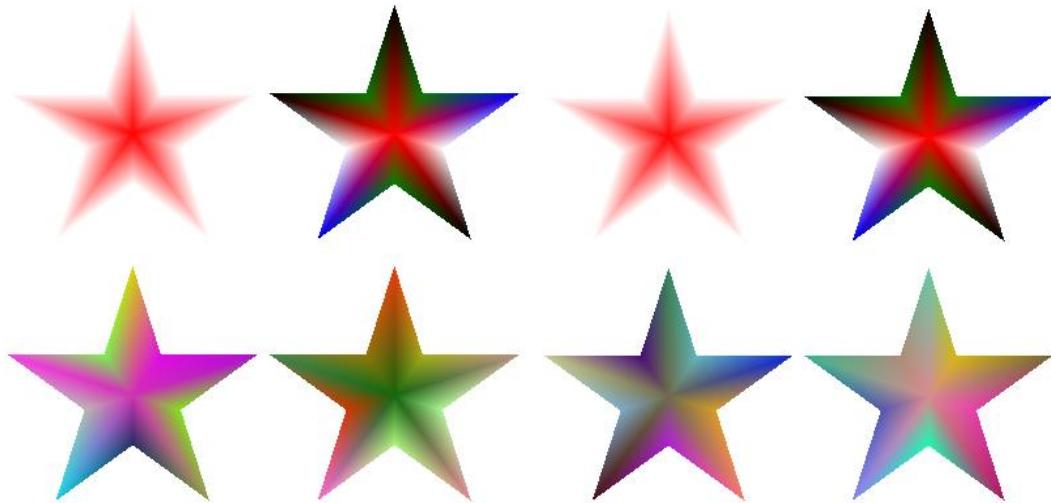
其中，路径刷的中心色和路径点色，缺省都为背景色（白色）。

例如：（用路径刷画五角星）

```
void FillPentacle(HDC hdc) {
    INT count = 10;
    Point points[] = {Point(100, 0), Point(122, 69), Point(195, 69), Point(137, 111), Point(159, 181),
                      Point(100, 138), Point(41, 181), Point(63, 111), Point(5, 69), Point(78, 69)};
    GraphicsPath path;
    path.AddPolygon(points, count);
    Graphics graph(hdc);
    PathGradientBrush pgBrush(&path);
    pgBrush.SetCenterColor(Color::Red);
    //pgBrush.SetCenterColor(Color::Green);
    graph.FillPath(&pgBrush, &path);
    Color cols[] = {Color::Black, Color::Green, Color::Blue, Color::White, Color::Black,
                    Color::Green, Color::Blue, Color::White, Color::Black, Color::Green};
    /*Color cols[] = {Color::Cyan, Color::Aqua, Color::Blue, Color::Chartreuse, Color::Coral,
                    Color::CadetBlue, Color::HotPink, Color::Turquoise, Color::LightSkyBlue, Color::DeepPink};*/
    pgBrush.SetSurroundColors(cols, &count);
    graph.TranslateTransform(200.0f, 0);
    graph.FillPath(&pgBrush, &path);
    for (int i = 0; i < count; i++) cols[i] = Color(rand() % 255, rand() % 255, rand() % 255);
    pgBrush.SetSurroundColors(cols, &count);
    pgBrush.SetCenterColor(Color(rand() % 255, rand() % 255, rand() % 255));
    graph.TranslateTransform(-200.0f, 200.0f);
    graph.FillPath(&pgBrush, &path);
    for (int i = 0; i < count; i++) cols[i] = Color(rand() % 255, rand() % 255, rand() % 255);
    pgBrush.SetSurroundColors(cols, &count);
```

```
pgBrush.SetCenterColor(Color(rand() % 255, rand() % 255, rand() % 255));  
graph.TranslateTransform(200.0f, 0.0f);  
graph.FillPath(&pgBrush, &path);  
}
```

输出结果为（由于下排的两个五角星，使用的是随机色，所以每次刷新时，颜色都不一样）



路径刷五角星（右上为红心和黑、绿、蓝、白边点，下排为随机色）



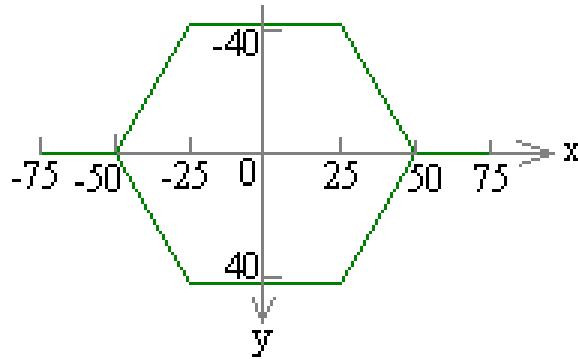
路径刷五角星（右上为白心和各色边点，下排为随机色）

如果将边点全改成红色、绿色或蓝色，中心改为白色，则输出效果为：



路径刷五角星（中心白色，边点分别为红绿蓝）

又例如：（用路径刷画平铺六边形）



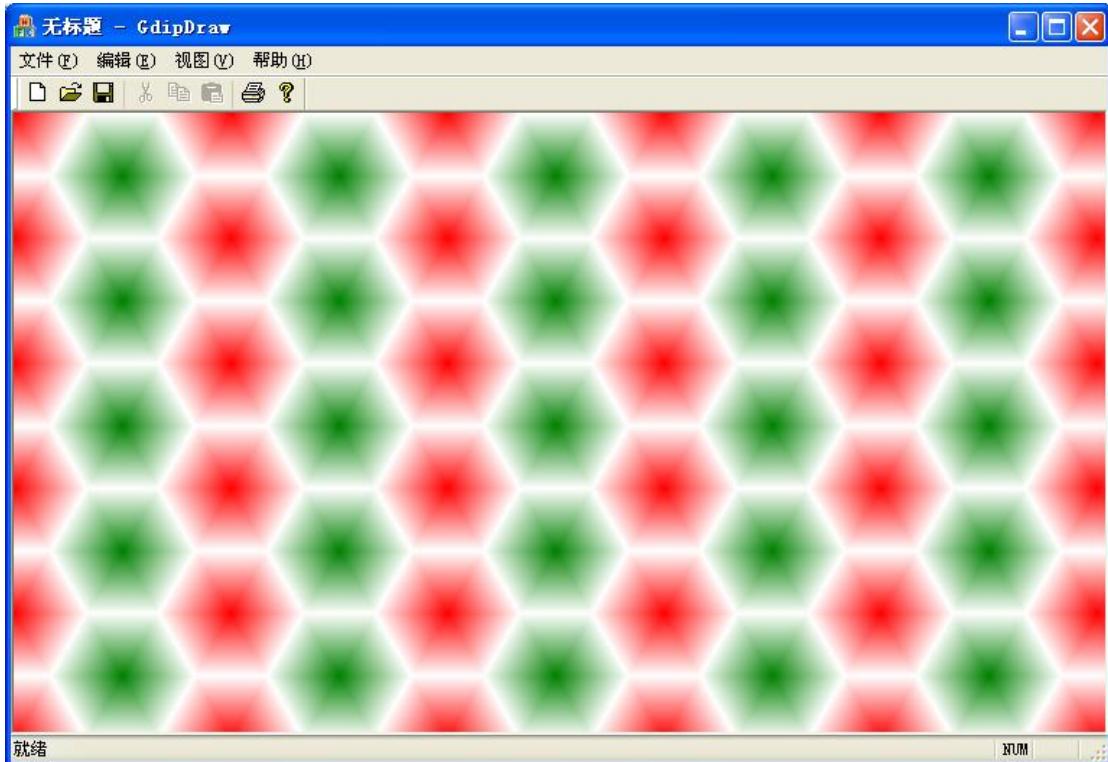
六边形坐标

```
#include <math.h>
void FillHexagon(HDC hdc, RectF &rect) // 用路径刷画平铺六边形
{
    float radian = 3.1415926f / 180.0f;
    float l = 50.0f, fh = l * sin(60.0f * radian);
    PointF pts[] = {PointF(50.0f, 0.0f), PointF(50.0f * 1.5f, 0.0f), PointF(50.0f, 0.0f),
                    PointF(50.0f / 2.0f, -fh), PointF(-50.0f / 2.0f, -fh),
                    PointF(-50.0f, 0.0f), PointF(-50.0f * 1.5f, 0.0f), PointF(-50.0f, 0.0f),
                    PointF(-50.0f / 2.0f, fh), PointF(50.0f / 2.0f, fh)};
    PathGradientBrush pgBrush(pts, 10);
    pgBrush.SetWrapMode(WrapModeTile);
    Graphics graph(hdc);
    pgBrush.SetCenterColor(Color::Red);
    graph.FillRectangle(&pgBrush, rect);
    graph.TranslateTransform(75.0f, fh);
    pgBrush.SetCenterColor(Color::Green);
    graph.FillRectangle(&pgBrush, -75.0f, -fh, rect.Width, rect.Height);
}
void CGdipDrawView::OnDraw(CDC* pDC) {
    .....
    RECT srect;
    GetClientRect(&srect);
```

```

    RectF rect(0.0f, 0.0f, REAL(srect.right), REAL(srect.bottom));
    FillHexagon(pDC->m_hDC, rect); // 用路径刷画平铺六边形
}

```



路径刷平铺六边形

5. 区域

区域（region）由若干几何形状所构成的一种封闭图形，主要用于复杂图形的绘制、图形输出的剪裁和鼠标击中的测试。最简单也是最常用的区域是矩形，其次是椭圆和多边形以及它们的组合。这些也正是 GDI 所支持的区域类型。

GDI+中的区域是一种显示表面的范围（an area of the display surface），可以是任意形状（的图形的组合），边界一般为路径。除了上面所讲的矩形、椭圆、多边形之外，其边界还可以含直线、折线、弧、贝塞尔曲线和样条曲线等开图形，其内容还可以包含饼、闭曲线等闭图形。

在 GDI+中，区域所对应的类是 Region，它是一个独立的类（没有基类，也没有派生类）。但是它又若干相关的类，如各种图形类和图形路径类等。

1) 构造函数

Region 类有 6 个构造函数：

```

Region(VOID); // 创建一个空区域
Region(const Rect &rect); // 创建一个整数型矩形区域
Region(const RectF &rect); // 创建一个浮点数型矩形区域
Region(const GraphicsPath *path); // 由图形路径来创建区域

```

```
Region(const BYTE *regionData, INT size); // 由（另一）区域的数据构造区域  
Region(HRGN hRgn); // 由 GDI 的区域句柄构造区域  
其中，创建矩形区域最简单，由路径创建区域最常用。
```

2) 其他函数

区域类的其他函数有：

```
// 克隆（复制）  
Region *Clone(VOID);  
// 判断本区域是否与另一区域 region 相等， g 用于坐标计算：  
BOOL Equals(const Region *region, const Graphics *g) const; // 相等  
// 空区域  
BOOL IsEmpty(const Graphics *g) const; // 判断区域是否为空 (g 也用于坐标计算)  
Status MakeEmpty(VOID); // 清空区域  
// 无限区域  
BOOL IsInfinite(const Graphics *g) const; // 判断区域是否无限 (g 也用于坐标计算)  
Status MakeInfinite(VOID); // 使区域成为无限的  
// 获取外接（包）矩形  
Status GetBounds(Rect *rect, const Graphics *g) const; // g 也用于坐标计算  
Status GetBounds(RectF *rect, const Graphics *g) const;  
// 获取区域的二进制数据，可以用于创建其他区域：  
UINT GetDataSize(VOID) const; // 获取区域数据的字节大小  
Status GetData(BYTE *buffer, UINT bufferSize, UINT *sizeFilled = NULL) const;  
// 获取区域的矩形逼近表示  
UINT GetRegionScansCount(const Matrix *matrix) const;  
Status GetRegionScans(const Matrix *matrix, Rect *rects, INT *count) const;  
Status GetRegionScans(const Matrix *matrix, RectF *rects, INT *count) const;  
// 平移区域  
Status Translate(INT dx, INT dy);  
Status Translate(REAL dx, REAL dy);  
// 变换区域  
Status Transform(const Matrix *matrix);  
// 获取最后一次区域操作的状态  
Status GetLastStatus(VOID); // 成功返回 Ok  
// 获取区域所对应的 GDI 区域的句柄  
HRGN GetHRGN(const Graphics *g) const; // g 也用于坐标计算  
还有集合运算和区域测试函数，将在后面分别介绍。
```

3) 区域的集合运算

- 并（union）

```
Status Union(const Region *region); // 与另一区域的并  
Status Union(const Rect &rect); // 与一整数型矩形的并
```

- Status Union(const Rect &rect); // 与一浮点数型矩形的并
 Status Union(const GraphicsPath *path); // 与一路径的并
- 交 (intersect)
 Status Intersect(const Region *region); // 与另一区域的交
 Status Intersect(const Rect &rect); // 与一整数型矩形的交
 Status Intersect(const Rect &rect); // 与一浮点数型矩形的交
 Status Intersect(const GraphicsPath *path); // 与一路径的交
- 余 (exclude)
 Status Exclude(const Region *region); // 与另一区域的余
 Status Exclude(const Rect &rect); // 与一整数型矩形的余
 Status Exclude(const Rect &rect); // 与一浮点数型矩形的余
 Status Exclude(const GraphicsPath *path); // 与一路径的余
- 补 (complement)
 Status Complement(const Region *region); // 与另一区域的补
 Status Complement(const Rect &rect); // 与一整数型矩形的补
 Status Complement(const Rect &rect); // 与一浮点数型矩形的补
 Status Complement(const GraphicsPath *path); // 与一路径的补
- 异或 (xor)
 Status Xor(const Region *region); // 与另一区域的异或
 Status Xor(const Rect &rect); // 与一整数型矩形的异或
 Status Xor(const Rect &rect); // 与一浮点数型矩形的异或
 Status Xor(const GraphicsPath *path); // 与一路径的异或

例如：

```

SolidBrush textBrush(Color::Black);
Font font(L"宋体", 10.5);
StringFormat stringFormat;
stringFormat.SetAlignment(StringAlignmentCenter);
Point points[] = {Point(100, 0), Point(122, 69), Point(195, 69), Point(137, 111),
    Point(159, 181), Point(100, 138), Point(41, 181), Point(63, 111), Point(5, 69),
    Point(78, 69)};
GraphicsPath path;
path.AddPolygon(points, 10);
Region *pRgn, rgnRect(Rect(0, 50, 200, 80));
SolidBrush *pBrush, gbrush(Color::Green), tbrush(Color::Turquoise);
CString strs[] = {L"原集", L"并集", L"交集", L"余集", L"补集", L"异或集"};
PointF pts[] = {PointF(0.0f, 0.0f), PointF(210.0f, 0.0f), PointF(210.0f, 0.0f),
    PointF(-420.0f, 220.0f), PointF(210.0f, 0.0f), PointF(210.0f, 0.0f)};
Graphics graph(pDC->m_hDC);
for (int i = 0; i < 6; i++) {
    pRgn = new Region(&path);
    switch (i) {
        case 1: pRgn->Union(&rgnRect); break;
        case 2: pRgn->Intersect(&rgnRect); break;
    }
}

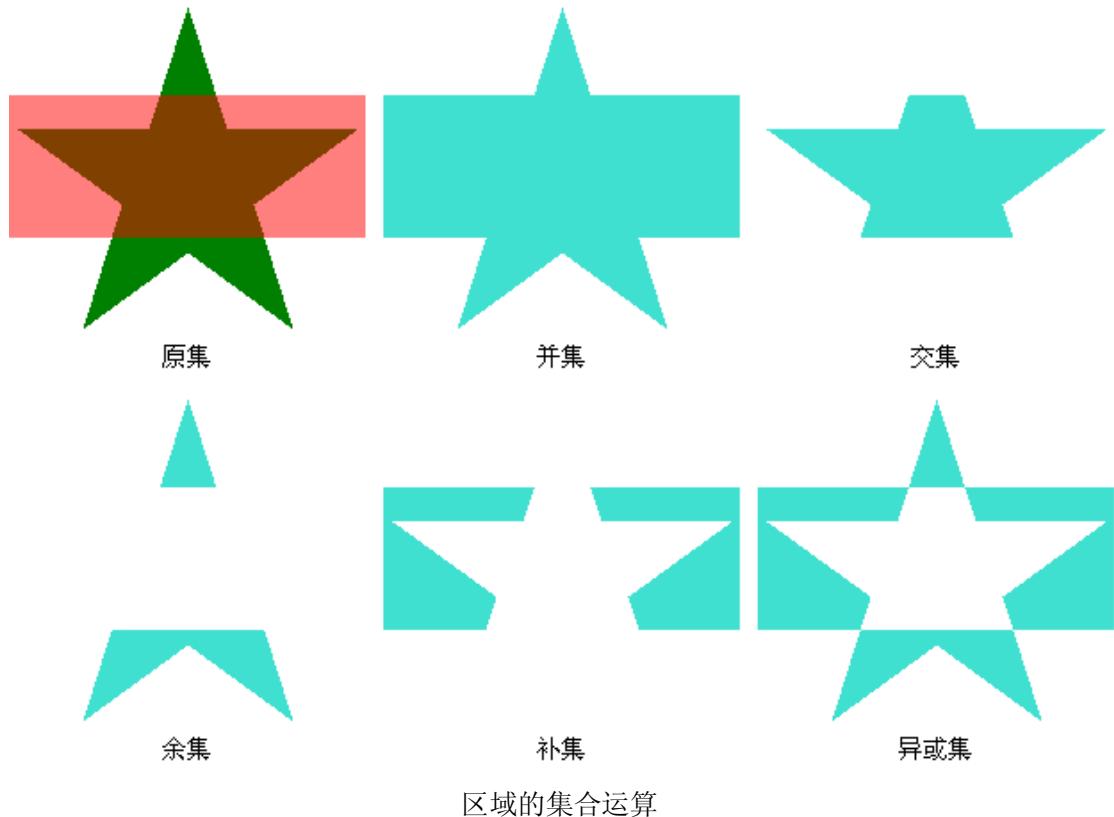
```

```

        case 3: pRgn->Exclude(&rgnRect); break;
        case 4: pRgn->Complement(&rgnRect); break;
        case 5: pRgn->Xor(&rgnRect); break;
    }
    if (i == 0) pBrush = &gbrush; else pBrush = &tbrush;
    graph.TranslateTransform(pts[i].X, pts[i].Y);
    graph.FillRegion(pBrush, pRgn);
    if (i == 0) graph.FillRegion(&SolidBrush (Color(128, 255, 0, 0)), &rgnRect);
    graph.DrawString(strs[i], -1, &font, PointF(100, 190), &stringFormat, &textBrush);
    delete pRgn;
}

```

输出结果为：



区域的集合运算

有例如：（图像不能太大，背景必须为单色）

```

Bitmap bmp(L"米老鼠和唐老鸭.bmp");
Graphics graph(pDC->m_hDC);
graph.DrawImage(&bmp, 0, 0);
int w = bmp.GetWidth(), h = bmp.GetHeight();
Region rgn(Rect(0, 0, w, h));
Color col, col0;
bmp.GetPixel(0, 0, &col0); // 获取背景色
ARGB argb0 = col0.GetValue();

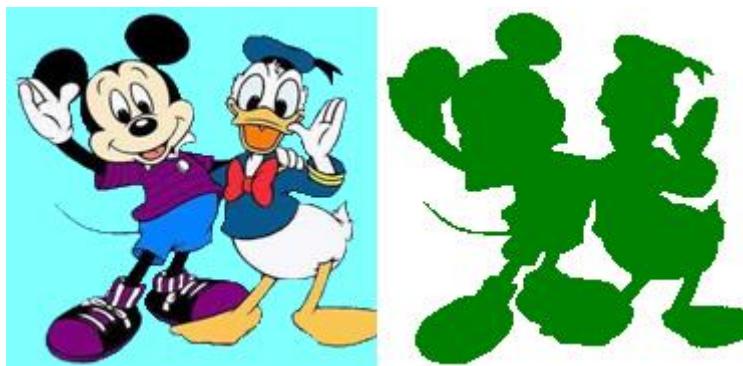
```

```

for (int i = 0; i < w; i++) {
    for (int j = 0; j < h; j++) {
        bmp.GetPixel(i, j, &col);
        if (col.GetValue() == argb0) {
            Region pixelRgn(Rect(i, j, 1, 1)); // 单像素矩形
            rgn.Exclude(&pixelRgn); // 挖去
        }
    }
}
graph.TranslateTransform(REAL(w), 0.0f);
graph.FillRegion(&SolidBrush(Color::Green), &rgn); // 容易溢出

```

输出结果为：



为了防止调用 FillRegion 函数造成堆栈溢出，必须减少区域的复杂度（用前面的方法，一个 180*180 的小图象所产生的抠像区域，要占 100KB 以上的存储空间）。解决办法之一是，不再每次去一个点，而是一次去一个（垂）直线段，则复杂度会减少一到两个数量级。从而可以避免溢出。

例如：

```

Bitmap bmp(L"米老鼠和唐老鸭 2.bmp");
Graphics graph(pDC->m_hDC);
graph.DrawImage(&bmp, 0, 0);
int w = bmp.GetWidth(), h = bmp.GetHeight();
Region rgn(Rect(0, 0, w, h));
Color col, col0;
bmp.GetPixel(0, 0, &col0); // 获取背景色
ARGB argb0 = col0.GetValue();
int j0, dj;
bool begin = false;
for (int i = 0; i < w; i++) {
    for (int j = 0; j < h; j++) {
        bmp.GetPixel(i, j, &col);
        if (col.GetValue() == argb0) {
            if (!begin) {
                j0 = j;

```

```

begin = true;
}
}
else if (begin) {
    dj = j - j0;
    Region pixelRgn(Rect(i, j0, 1, dj)); // 1*dj 像素矩形
    rgn.Exclude(&pixelRgn); // 挖去
    begin = false;
}
}
if (begin) {
    dj = h - j0;
    Region pixelRgn(Rect(i, j0, 1, dj)); // 1*dj 像素矩形
    rgn.Exclude(&pixelRgn); // 挖去
    begin = false;
}
}
graph.TranslateTransform(REAL(w), 0.0f);
graph.FillRegion(&SolidBrush(Color::Green), &rgn); // 不易溢出

```

输出结果为：



4) 击中测试

```

// 点是否在区域内 (g 也用于坐标计算)
BOOL IsVisible(INT x, INT y, const Graphics *g) const; // 整数点坐标
BOOL IsVisible(const Point &point, const Graphics *g) const; // 整数点
BOOL IsVisible(REAL x, REAL y, const Graphics *g) const; // 浮点数点坐标
BOOL IsVisible(const PointF &point, const Graphics *g) const; // 浮点数点

```

```

// 矩形是否在区域内 (g 也用于坐标计算)
// 整数左上角坐标与高宽
BOOL IsVisible(INT x, INT y, INT width, INT height, const Graphics *g) const;
BOOL IsVisible(const Rect &rect, const Graphics *g) const; // 整数矩阵
// 浮点数左上角坐标与高宽
BOOL IsVisible(REAL x, REAL y, REAL width, REAL height, const Graphics *g) const;
BOOL IsVisible(const RectF &rect, const Graphics *g) const; // 浮点矩阵

例如，可由前面所讲的文字路径（空心字符串）来构造文字区域，并进行鼠标点是否区域内的测试。代码如下：

// 在视图类中定义区域指针类变量
class CGdipDrawView : public CView {
    .....
    Region *pRgn;
    .....
}

// 在视图类的初始化函数中，创建字符串路径和区域
void CGdipDrawView::OnInitialUpdate() {
    CView::OnInitialUpdate();
    FontFamily ff(L"隶书");
    GraphicsPath path;
    path.AddString(L"文字区域", -1, &ff, FontStyleRegular, 150, Point(0, 0), NULL);
    pRgn = new Region(&path);
}

// 在视图类的 OnDraw 函数中绘制
void CGdipDrawView::OnDraw(CDC* pDC) {
    .....
    Graphics graph(pDC->m_hDC);
    graph.FillRegion(&SolidBrush(Color::Green), pRgn);
}

// 在视图类的左鼠标键松开的消息响应函数中，判断当前鼠标位置是否在文字区域中
void CGdipDrawView::OnLButtonUp(UINT nFlags, CPoint point) {
    // TODO: 在此添加消息处理程序代码和/或调用默认值
    if(pRgn->IsVisible(point.x, point.y)) MessageBox(L"鼠标在区域内");
    else MessageBox(L"鼠标在区域外");
}

// 在视图类的析构函数中，删除所创建的区域对象
CGdipDrawView::~CGdipDrawView() {
    delete pRgn;
}

```

输出结果如：



6. 变换

变换 (transform) 是 GDI+新增加的强大功能，包括图形对象的简单变换和基于矩阵的坐标变换、图形变换、图像变换、色彩变换、路径变换和区域变换等。

1) 图形对象变换

GDI+的核心——图形类 `Graphics`，提供了 3 个成员函数，可以对其所绘制的图形进行平移 (`translate`)、旋转 (`rotate`) 和伸缩 (`scale` 比例尺/缩放) 等基本的图形变换：(与纹理刷类中的对应函数的原型是一样的)

```
Status TranslateTransform(REAL dx, REAL dy, MatrixOrder order = MatrixOrderPrepend);
```

```
Status RotateTransform(REAL angle, MatrixOrder order = MatrixOrderPrepend);
```

```
Status ScaleTransform(REAL sx, REAL sy, MatrixOrder order = MatrixOrderPrepend);
```

其中的最后一个输入参数为矩阵相乘的顺序，取值为矩阵顺序枚举类型 `MatrixOrder` 中的符号常量，缺省值都为 `MatrixOrderAppend` (左乘)：

```
typedef enum {
    MatrixOrderPrepend = 0, // 矩阵左乘 (预先序, 前置)
    MatrixOrderAppend = 1 // 矩阵右乘 (追加序, 后缀)
} MatrixOrder
```

因为这些变换都可以用矩阵表示，而且与图形对象已经设置的现有变换矩阵要进行合成 (相当于两个变换矩阵进行乘法运算)。

在图形对象的这三种基本变换中，最常用的是第一种——平移变换。我们在前面曾多次使用，避免了重复定义 (有坐标平移的) 绘图区域的麻烦。

(1) 平移变换 `TranslateTransform`

平移变换将所绘制图形的坐标(x, y)全部平移一个增量(dx, dy)：

```
Status TranslateTransform(REAL dx, REAL dy, MatrixOrder order = MatrixOrderPrepend);
```

即：

$$\begin{aligned}x' &= x + d_x \\y' &= y + d_y\end{aligned}$$

相当于进行如下矩阵变换：

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} d_x \\ d_y \end{pmatrix}$$

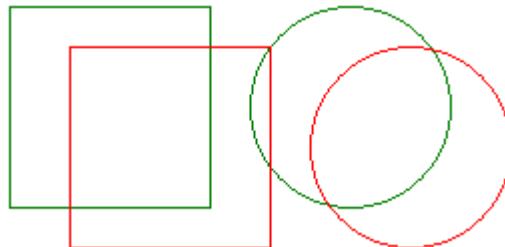
为了与旋转和伸缩等其他变换合成，可以改写为齐次坐标形式：

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

例如：

```
Graphics graph(pDC->m_hDC);
RedrawWindows(); // 强制窗口重画
Rect rectSquare(10, 10, 100, 100), rectCircle(120, 10, 100, 100);
graph.DrawRectangle(&Pen(Color::Green), rectSquare);
graph.DrawEllipse(&Pen(Color::Green), rectCircle);
graph.TranslateTransform(30, 20);
graph.DrawRectangle(&Pen(Color::Red), rectSquare);
graph.DrawEllipse(&Pen(Color::Red), rectCircle);
```

输出结果为：



平移(30, 20)

(2) 旋转变换 RotateTransform

旋转变换将所绘制图形的坐标(x, y)全部（相对于坐标原点[左上角]）旋转一个角度 $\alpha = \text{angle}$ (单位为度)：

Status **RotateTransform**(REAL angle, MatrixOrder order = MatrixOrderPrepend);

即：

$$\begin{aligned} x' &= x * \cos \alpha - y * \sin \alpha \\ y' &= x * \sin \alpha + y * \cos \alpha \end{aligned}$$

相当于进行如下矩阵变换：

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

为了与平移和伸缩等其他变换合成，也可以改写为齐次坐标形式：

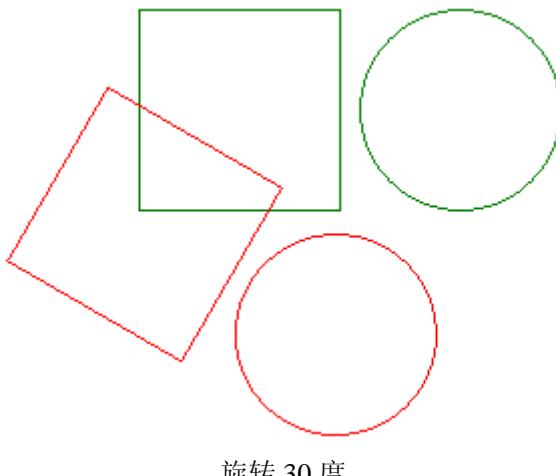
$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

注意，旋转都是相对于坐标原点来进行的，为了使你的图形可以自由旋转，必须相对于原点来绘图。

例如：

```
Graphics graph(pDC->m_hDC);
Rect rectSquare(80, 10, 100, 100), rectCircle(190, 10, 100, 100);
graph.DrawRectangle(&Pen(Color::Green), rectSquare);
graph.DrawEllipse(&Pen(Color::Green), rectCircle);
graph.RotateTransform(30.0f);
graph.DrawRectangle(&Pen(Color::Red), rectSquare);
graph.DrawEllipse(&Pen(Color::Red), rectCircle);
```

输出结果为：

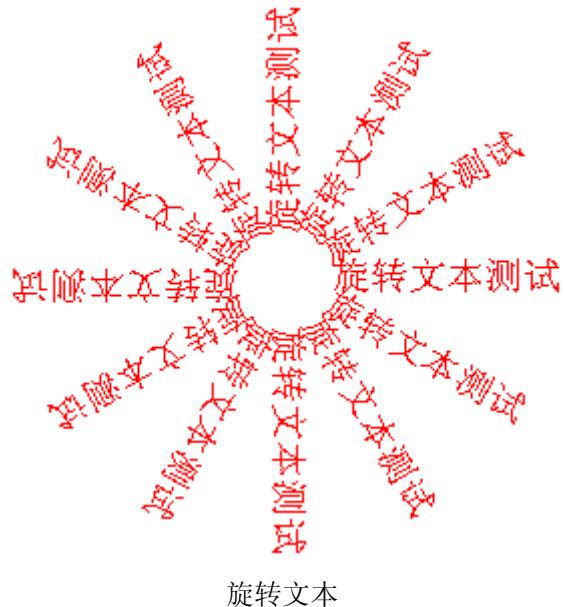


又例如：（旋转文本）

```
Graphics graph(pDC->m_hDC);
CRect crect;
GetClientRect(&crect);
PointF cp(REAL(crect.CenterPoint().x), REAL(crect.CenterPoint().y));
Font font(L"宋体", 14);
SolidBrush textBrush(Color::Red);
StringFormat stringFormat;
stringFormat.SetLineAlignment(StringAlignmentCenter);
graph.TranslateTransform(cp.X, cp.Y); // 旋转中心
for (int i = 0; i < 360; i += 30) {
    graph.RotateTransform(REAL(i)); // 旋转
    graph.DrawString(L"旋转文本测试", -1, &font, PointF(20.0f, 0.0f),
        &stringFormat, &textBrush);
    graph.RotateTransform(REAL(-i)); // 还原
}
```

}

输出结果为：



(3) 伸缩变换 ScaleTransform

伸缩变换将所绘制图形的形状在 x 和 y 方向上按指定比例 sx 和 sy 进行缩放：

Status **ScaleTransform**(REAL sx, REAL sy, MatrixOrder order = MatrixOrderPrepend);

即：

$$\begin{aligned}x' &= s_x * x \\y' &= s_y * y\end{aligned}$$

相当于进行如下矩阵变换：

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

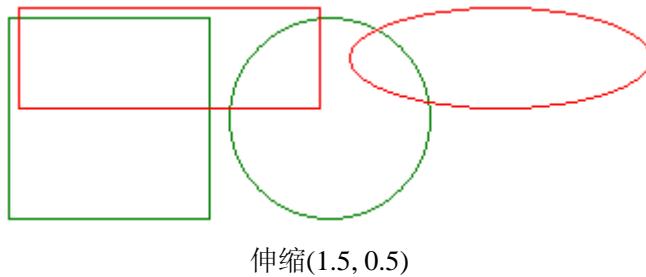
为了与平移和旋转等其他变换合成，也可以改写为齐次坐标形式：

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

例如：

```
Graphics graph(pDC->m_hDC);
Rect rectSquare(10, 10, 100, 100), rectCircle(120, 10, 100, 100);
graph.DrawRectangle(&Pen(Color::Green), rectSquare);
graph.DrawEllipse(&Pen(Color::Green), rectCircle);
graph.ScaleTransform(1.5f, 0.5f);
graph.DrawRectangle(&Pen(Color::Red), rectSquare);
graph.DrawEllipse(&Pen(Color::Red), rectCircle);
```

输出结果为：



还可以用负值参数来调用伸缩函数，达到镜像输出图形的效果。

例如：（镜像文本）

```
Graphics graph(pDC->m_hDC);
Font font(L"宋体", 50, FontStyleRegular, UnitPixel),
    ifont(L"宋体", 50, FontStyleItalic, UnitPixel);
SolidBrush textBrush(Color::Blue), mirrorBrush(Color::Gray);
// 普通文本输出
CString str(L"镜像文本");
graph.DrawString(str, str.GetLength(), &font, PointF(200.0f, 10.0f), &textBrush);
// 垂直镜像文本输出（上下倒置）
graph.ScaleTransform(1, -1);
graph.TranslateTransform(0, -120);
graph.DrawString(str, str.GetLength(), &ifont, PointF(200.0f, 10.0f), &mirrorBrush);
// 水平镜像文本输出（左右倒置）
graph.ResetTransform();
graph.ScaleTransform(-1, 1);
graph.TranslateTransform(-220, 0);
graph.DrawString(str, str.GetLength(), &font, PointF(0.0f, 10.0f), &mirrorBrush);
// 垂直+水平镜像文本输出（上下左右全倒置）
graph.ResetTransform();
graph.ScaleTransform(-1, -1);
graph.TranslateTransform(-220, -120);
graph.DrawString(str, str.GetLength(), &ifont, PointF(0.0f, 10.0f), &mirrorBrush);
```

输出结果为：



又例如：（镜像图像）

```
Graphics graph(pDC->m_hDC);
Image img(L"金泰熙.bmp");
```

```
REAL w = REAL(img.GetWidth()), h = REAL(img.GetHeight());
graph.DrawImage(&img, PointF(w, 0));
graph.ScaleTransform(1, -1);
graph.TranslateTransform(0, -2 * h);
graph.DrawImage(&img, PointF(w, 0));
graph.ResetTransform();
graph.ScaleTransform(-1, 1);
graph.TranslateTransform(-w, 0);
graph.DrawImage(&img, PointF(0, 0));
graph.ResetTransform();
graph.ScaleTransform(-1, -1);
graph.TranslateTransform(-w, -2 * h);
graph.DrawImage(&img, PointF(0, 0));
```

输出结果为：



镜像图像

(4) 混合变换

也可以将者 3 种变换结合起来，形成混合变换。注意变换的顺序，顺序不同，结果可能页不一样。例如：

- 先旋转后伸缩

相当于：

$$x' = x \cos \alpha - y \sin \alpha$$

$$y' = x \sin \alpha + y \cos \alpha$$

$$x'' = s_x * x'$$

$$y'' = s_y * y'$$

即：

$$x'' = s_x * x' = s_x * (x \cos \alpha - y \sin \alpha) = s_x * x \cos \alpha - s_x * y \sin \alpha$$

$$y'' = s_y * y' = s_y * (x \sin \alpha + y \cos \alpha) = s_y * x \sin \alpha + s_y * y \cos \alpha$$

或

$$\begin{pmatrix} x'' \\ y'' \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \begin{pmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

$$= \begin{pmatrix} s_x \cos \alpha & -s_x \sin \alpha \\ s_y \sin \alpha & s_y \cos \alpha \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

也可以改写为齐次坐标形式：

$$\begin{pmatrix} x'' \\ y'' \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

$$= \begin{pmatrix} s_x \cos \alpha & -s_x \sin \alpha & 0 \\ s_y \sin \alpha & s_y \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

上面用的都是一般的列向量形式，下面是 GDI+所采用的行向量形式：

$$(x'' \ y'' \ 1) = (x' \ y' \ 1) \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= (x \ y \ 1) \begin{pmatrix} \cos \alpha & \sin \alpha & 0 \\ -\sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= (x \ y \ 1) \begin{pmatrix} s_x \cos \alpha & s_y \sin \alpha & 0 \\ -s_x \sin \alpha & s_y \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

例如：

```
Graphics graph(pDC->m_hDC);
```

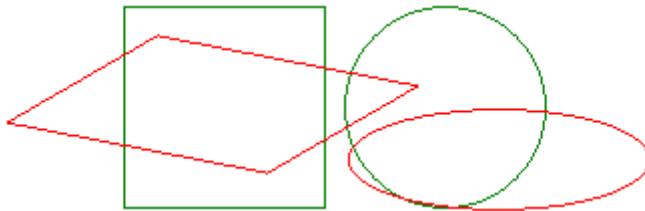
```
Rect rectSquare(30, 10, 100, 100), rectCircle(140, 10, 100, 100);
```

```

graph.DrawRectangle(&Pen(Color::Green), rectSquare);
graph.DrawEllipse(&Pen(Color::Green), rectCircle);
graph.ScaleTransform(1.5f, 0.5f);
graph.RotateTransform(30.0f); // 左乘, 先变换
graph.DrawRectangle(&Pen(Color::Red), rectSquare);
graph.DrawEllipse(&Pen(Color::Red), rectCircle);

```

输出结果为：



旋转 30 度后再伸缩(1.5, 0.5)

● 先伸缩后旋转

相当于：

$$x' = s_x * x$$

$$y' = s_y * y$$

$$x'' = x' * \cos \alpha - y' * \sin \alpha$$

$$y'' = x' * \sin \alpha + y' * \cos \alpha$$

即：

$$x'' = x' * \cos \alpha - y' * \sin \alpha = (s_x * x) * \cos \alpha - (s_y * y) * \sin \alpha = s_x * x * \cos \alpha - s_y * y * \sin \alpha$$

$$y'' = x' * \sin \alpha + y' * \cos \alpha = (s_x * x) * \sin \alpha + (s_y * y) * \cos \alpha = s_x * x * \sin \alpha + s_y * y * \cos \alpha$$

或

$$\begin{aligned} \begin{pmatrix} x'' \\ y'' \\ 1 \end{pmatrix} &= \begin{pmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} s_x \cos \alpha & -s_y \sin \alpha & 0 \\ s_x \sin \alpha & s_y \cos \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \end{aligned}$$

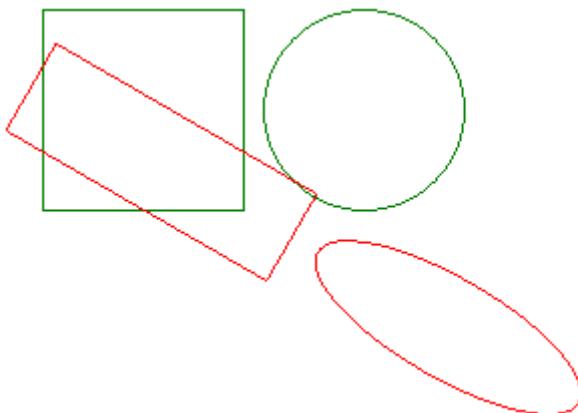
例如：

```

Graphics graph(pDC->m_hDC);
Rect rectSquare(80, 10, 100, 100), rectCircle(190, 10, 100, 100);
graph.DrawRectangle(&Pen(Color::Green), rectSquare);
graph.DrawEllipse(&Pen(Color::Green), rectCircle);
graph.RotateTransform(30.0f); // MatrixOrderPrepend
graph.ScaleTransform(1.5f, 0.5f); // MatrixOrderPrepend 左乘, 先变换
graph.DrawRectangle(&Pen(Color::Red), rectSquare);
graph.DrawEllipse(&Pen(Color::Red), rectCircle);

```

输出结果为：



伸缩(1.5, 0.5)后再旋转 30 度

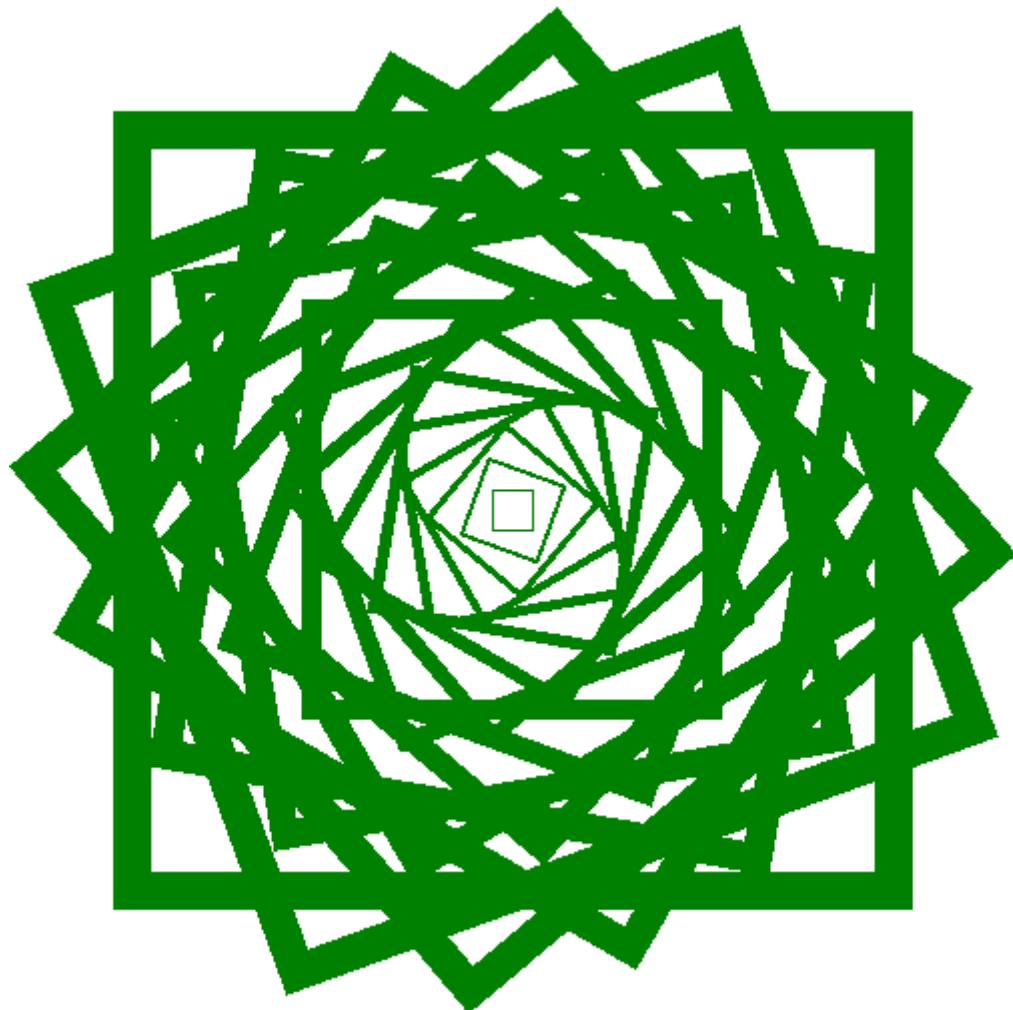
上面的红色代码部分，也可以替换为：

```
graph.ScaleTransform(1.5f, 0.5f); // MatrixOrderPrepend  
graph.RotateTransform(30.0f, MatrixOrderAppend); // 右乘，后变换
```

又例如：（旋转正方形）

```
Graphics graph(pDC->m_hDC);  
CRect crect;  
GetClientRect(&crect); // 获取当前客户区矩形  
PointF cp(REAL(crect.CenterPoint().x), REAL(crect.CenterPoint().y)); // 中心点  
RectF rect(cp.X - 10.0f, cp.Y - 10.0f, 20.0f, 20.0f); // 原始矩形  
REAL scale = 1.0f; // 初始放大倍数  
//REAL scale = 0.0f, ds = 10.0f; // 用于矩形膨胀  
  
for (int i = 0; i <= 360; i += 20) { // 循环  
    graph.TranslateTransform(cp.X, cp.Y); // 设置旋转中心  
    //rect.Inflate(scale * ds, scale * ds); // 膨胀矩形  
    graph.ScaleTransform(scale, scale); // 放大  
    graph.RotateTransform(REAL(i)); // 旋转  
    graph.TranslateTransform(-cp.X, -cp.Y); // 还原坐标原点  
    graph.DrawRectangle(&Pen(Color::Green), rect); // 绘制矩形  
    Sleep(100); // 暂停 100 毫秒 (0.1 秒)  
    graph.ResetTransform(); // 重置所有变换 (还原为恒等变换)  
    //rect.Inflate(-scale * ds, -scale * ds); // 还原矩形原大小  
    scale += 1.0f; // 放大倍数加 1  
}
```

输出结果为：



旋转并放大矩形

其中的 SDK 的睡眠函数 Sleep:

```
VOID Sleep(DWORD dwMilliseconds);
```

让当前线程暂停执行 dwMilliseconds 毫秒。

在上例中，因为矩形的框线随矩形一起被放大，所以边框越来越粗。如果注释掉语句：

```
//REAL scale = 1.0f;
```

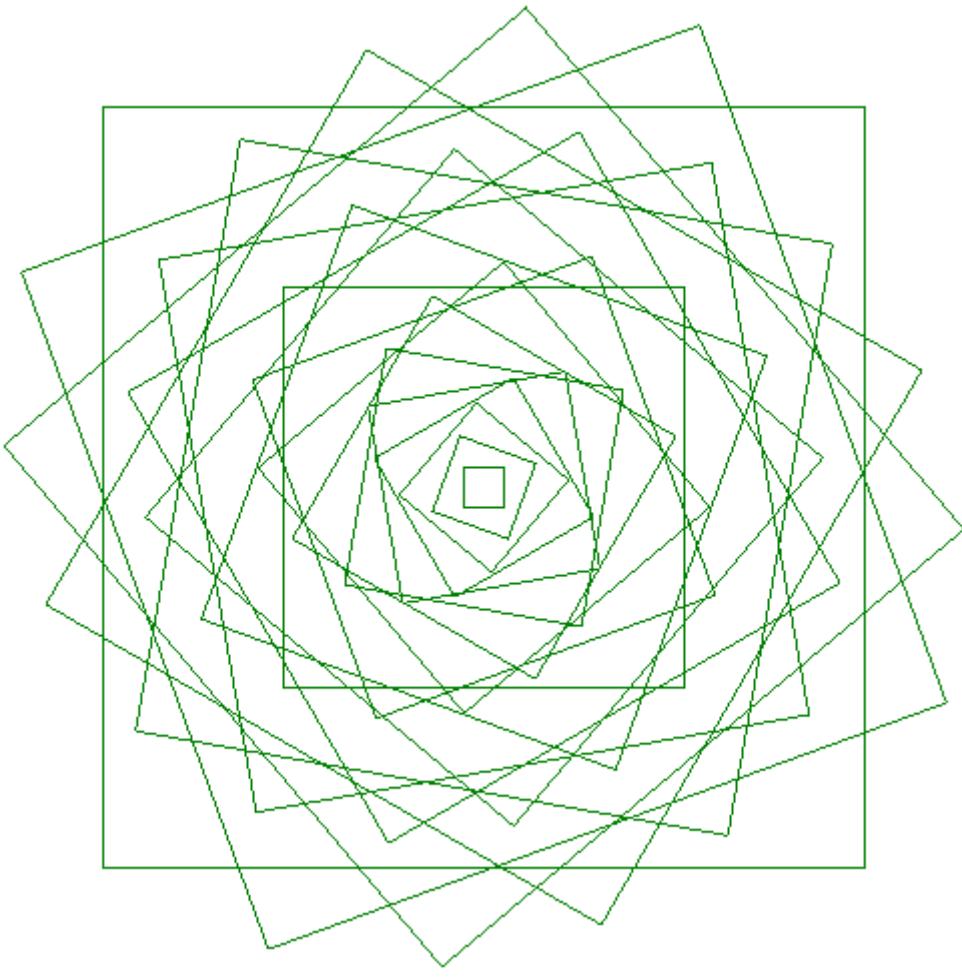
和

```
//graph.ScaleTransform(scale, scale);
```

并去掉原来的三个注释符，把对整个图形的放大改为只有矩形本身膨胀（inflate）：

```
rRect.Inflate(scale * ds, scale * ds);
```

则框线将保持单个像素粗，输出结果变为：



旋转并膨胀矩形

这里使用了矩形类 Rect 和 RectF 的成员函数 Inflate (膨胀):

```
VOID Inflate(INT dx, INT dy);
VOID Inflate(const Point& point);
VOID Inflate(REAL dx, REAL dy);
VOID Inflate(const PointF& point);
```

将原矩形的左右和上下分别膨胀 dx 和 dy 个逻辑单位 (缺省为像素), 矩形的中心不变。

(5) 矩阵变换

除了上面所讲的 3 种基本变换外, 图形类还可以设置复杂的矩阵变换。有关的成员函数为:

```
Status SetTransform(const Matrix *matrix); // 设置变换矩阵为 matrix
Status ResetTransform(VOID); // 设置变换矩阵为单位矩阵 (恒等变换) (还原)
```

例如:

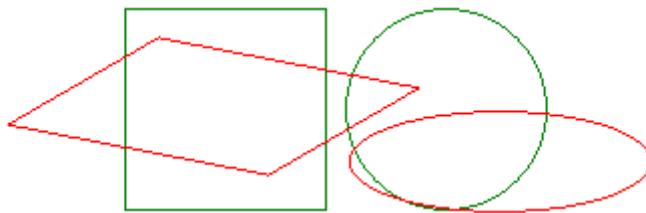
```
Graphics graph(pDC->m_hDC);
REAL radian = 3.1415926f / 180.0f;
Rect rectSquare(80/*30*/, 10, 100, 100), rectCircle(190/*140*/, 10, 100, 100);
REAL sx = 1.5f, sy = 0.5f, a = 30.0f;
```

```

REAL m11 = sx * cos(a * radian), m12 = sy /*-sx/* sin(a * radian),
m21 = -sx/*sy/* sin(a * radian), m22 = sy * cos(a * radian),
m31 = 0.0f, m32 = 0.0f;
Matrix m(m11, m12, m21, m22, m31, m32);
graph.DrawRectangle(&Pen(Color::Green), rectSquare);
graph.DrawEllipse(&Pen(Color::Green), rectCircle);
graph.SetTransform(&m);
graph.DrawRectangle(&Pen(Color::Red), rectSquare);
graph.DrawEllipse(&Pen(Color::Red), rectCircle);

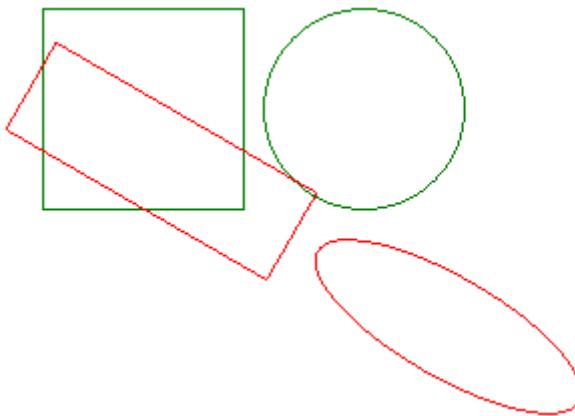
```

输出结果为：(相当于前面的先旋转后伸缩)



矩阵变换（旋转 30 度+伸缩(1.5, 0.5)）

若将上面 rectSquare 的 X 值改为 30、rectCircle 的 X 值改为 140、m12 中的 sy 改为 sx、m21 中的-sx 改为-sy，则输出结果为：(相当于前面的先伸缩后旋转)



矩阵变换（伸缩(1.5, 0.5)+旋转 30 度）

下面的小小节 2) 将介绍矩阵变换的详细内容。

2) 矩阵变换

在 GDI+中，矩阵变换所涉及的矩阵，由专门的矩阵类 Matrix 来表示。不过这里的矩阵，不仅可以用于图形变换，也可以用于图像、颜色、路径、区域等对象的变换。

GDI+中的 3*3 的矩阵变换（matrix transformation）对应于几何上的仿射变换（affine transformation）：

行向量形式（GDI+）：

$$\begin{pmatrix} x' & y' & 1 \end{pmatrix} = \begin{pmatrix} x & y & 1 \end{pmatrix} \begin{pmatrix} s_x \cos \alpha & s_x \sin \alpha & 0 \\ -s_y \sin \beta & s_y \cos \beta & 0 \\ d_x & d_y & 1 \end{pmatrix}$$

列向量形式(一般):

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} s_x \cos \alpha & -s_y \sin \beta & d_x \\ s_x \sin \alpha & s_y \cos \beta & d_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

相当于非齐次坐标形式:

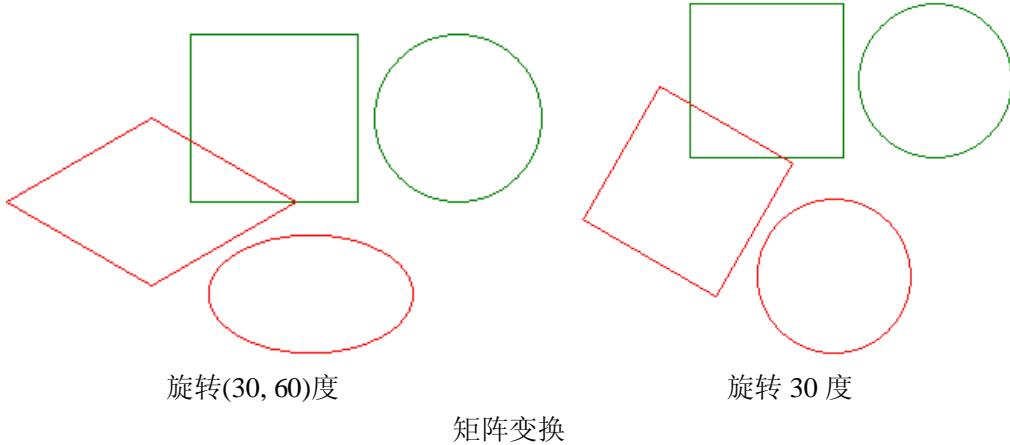
$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} s_x \cos \alpha & -s_y \sin \beta \\ s_x \sin \alpha & s_y \cos \beta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} d_x \\ d_y \end{pmatrix}$$

其中, s_x 、 α 、 d_x 和 s_y 、 β 、 d_y 分别为 x 和 y 方向的伸缩比例、旋转角度、平移量。(在 GDI+ 中, 一般取 $\alpha = \beta$)

例如, 只旋转不伸缩, 但是设 y 方向的旋转角度为 $b = 60$ 度与 x 方向的 $a = 30$ 度不同。
将上例中的矩阵元素计算部分改为:

```
Rect rectSquare(110, 10, 100, 100), rectCircle(220, 10, 100, 100);
REAL sx = 1.0f, sy = 1.0f, a = 30.0f, b = 60.0f;
REAL m11 = sx * cos(a * radian), m12 = sy * sin(a * radian),
      m21 = -sx * sin(b * radian), m22 = sy * cos(b * radian),
      m31 = 0.0f, m32 = 0.0f;
```

则输出结果为: (右图为将上面的 b 改回为 30 度的输出结果)



(1) 矩阵类 Matrix

矩阵类 Matrix 的构造函数有 4 个:

```
Matrix(VOID); // 单位矩阵
Matrix(const RectF &rect, const PointF *dstplg);
Matrix(const Rect &rect, const Point *dstplg);
Matrix(REAL m11, REAL m12, REAL m21, REAL m22, REAL dx, REAL dy);
```

其中:

- 单位矩阵:

$$M = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = M^T$$

- 变换矩阵:

$$M = \begin{pmatrix} rect.X & rect.Y & 0 \\ rect.Width & rect.Height & 0 \\ dstplg.X & dstplg.Y & 1 \end{pmatrix}, \quad M^T = \begin{pmatrix} rect.X & rect.Width & dstplg.X \\ rect.Y & rect.Height & dstplg.Y \\ 0 & 0 & 1 \end{pmatrix}$$

或:

$$M = \begin{pmatrix} m11 & m12 & 0 \\ m21 & m22 & 0 \\ dx & dy & 1 \end{pmatrix}, \quad M^T = \begin{pmatrix} m11 & m21 & dx \\ m12 & m22 & dy \\ 0 & 0 & 1 \end{pmatrix}$$

对应的矩阵变换（仿射变换）为:

行向量形式 (GDI+):

$$\begin{aligned} (x' \ y' \ 1) &= (x \ y \ 1)M \\ &= (x \ y \ 1) \begin{pmatrix} rect.X & rect.Y & 0 \\ rect.Width & rect.Height & 0 \\ dstplg.X & dstplg.Y & 1 \end{pmatrix} \\ &= (x \ y \ 1) \begin{pmatrix} m11 & m12 & 0 \\ m21 & m22 & 0 \\ dx & dy & 1 \end{pmatrix} \end{aligned}$$

列向量形式 (一般):

$$\begin{aligned} \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} &= M^T \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} rect.X & rect.Width & dstplg.X \\ rect.Y & rect.Height & dstplg.Y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} m11 & m21 & dx \\ m12 & m22 & dy \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \end{aligned}$$

实际上, GDI+中的矩阵并不是真正的 3*3 的 (似上面), 而只是 3*2 的 (但与上面 3*3

的矩阵等价), 剩下的最右一列(1)和 $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ 被省掉了:

$$\begin{aligned}
(x' & \quad y') = (x \quad y \quad 1) M_{3 \times 2} \\
&= (x \quad y \quad 1) \begin{pmatrix} rect.X & rect.Y \\ rect.Width & rect.Height \\ dstplg.X & dstplg.Y \end{pmatrix} \\
&= (x \quad y \quad 1) \begin{pmatrix} m11 & m12 \\ m21 & m22 \\ dx & dy \end{pmatrix} \\
&= (x \quad y \quad 1) \begin{pmatrix} s_x \cos \alpha & s_x \sin \alpha \\ -s_y \sin \beta & s_y \cos \beta \\ d_x & d_y \end{pmatrix}
\end{aligned}$$

其中：

$$\begin{aligned}
m11 &= s_x \cos \alpha, \quad m12 = s_x \sin \alpha \\
m21 &= -s_y \sin \beta, \quad m22 = s_y \cos \beta
\end{aligned}$$

对应的：

$$\begin{aligned}
s_x &= \sqrt{m11^2 + m12^2}, \quad \alpha = \operatorname{tg}^{-1} \frac{m12}{m11} \\
s_y &= \sqrt{m21^2 + m22^2}, \quad \beta = \operatorname{tg}^{-1} \left(-\frac{m21}{m22} \right)
\end{aligned}$$

(2) 矩阵函数

矩阵类 Matrix 提供了若干成员函数，用于获取矩阵信息以及进行各种矩阵运算：（按字母顺序排列）

```

Matrix *Clone(VOID); // 克隆
static BOOL Equals(const Matrix* matrix); // 相等
Status GetElements(REAL *m) const; // 获取元素数组 (m11~m32)
Status GetLastStatus(VOID); // 获取最后一次矩阵操作的状态
Status Invert(VOID); // 求逆 (将当前矩阵用其逆矩阵代替)
BOOL IsIdentity(VOID); // 判断是否为单位矩阵
BOOL IsInvertible(VOID); // 判断矩阵是否可逆
// 相乘 (将当前矩阵用与指定矩阵的乘积矩阵代替):
Status Multiply(const Matrix *matrix, MatrixOrder order = MatrixOrderPrepend);
REAL OffsetX(VOID); // x 偏移 (返回变换矩阵的 x 向位移 d_x)
REAL OffsetY(VOID); // y 偏移 (返回变换矩阵的 y 向位移 d_y)
Status Reset(VOID); // 重置 (用单位矩阵代替当前矩阵)
Status Rotate(REAL angle, MatrixOrder order = MatrixOrderPrepend); // 旋转
Status RotateAt(REAL angle, const PointF &center, // 相对于指定点旋转
                MatrixOrder order = MatrixOrderPrepend);
Status Scale(REAL scaleX, REAL scaleY, MatrixOrder order = MatrixOrderPrepend); // 伸缩

```

```

// 设置矩阵元素:
Status SetElements(REAL m11, REAL m12, REAL m21, REAL m22, REAL dx, REAL dy);
// 剪切 (将当前矩阵用与指定剪切矩阵的乘积矩阵代替):
Status Shear(REAL shearX, REAL shearY, MatrixOrder order = MatrixOrderPrepend);
Status TransformPoints(Point *pts, INT count = 1); // 变换整数点数组
Status TransformPoints(PointF *pts, INT count = 1); // 变换浮点数点数组
Status TransformVectors(Point *pts, INT count = 1); // 变换整数点向量
Status TransformVectors(PointF *pts, INT count = 1); // 变换浮点数点向量
Status Translate(REAL offsetX, REAL offsetY, MatrixOrder order = MatrixOrderPrepend); // 平移

```

(3) 剪切矩阵

函数 Shear (剪切/滑移) 的功能相当于投射变换, 变换矩阵为

$$S = \begin{pmatrix} 1 & shearY \\ shearX & 1 \\ 0 & 0 \end{pmatrix}$$

对应的坐标变换为:

$$\begin{aligned} (x' \quad y') &= (x \quad y \quad 1)S = (x \quad y \quad 1) \begin{pmatrix} 1 & shearY \\ shearX & 1 \\ 0 & 0 \end{pmatrix} \\ &= (x + y \cdot shearX \quad y + x \cdot shearY) \end{aligned}$$

例如:

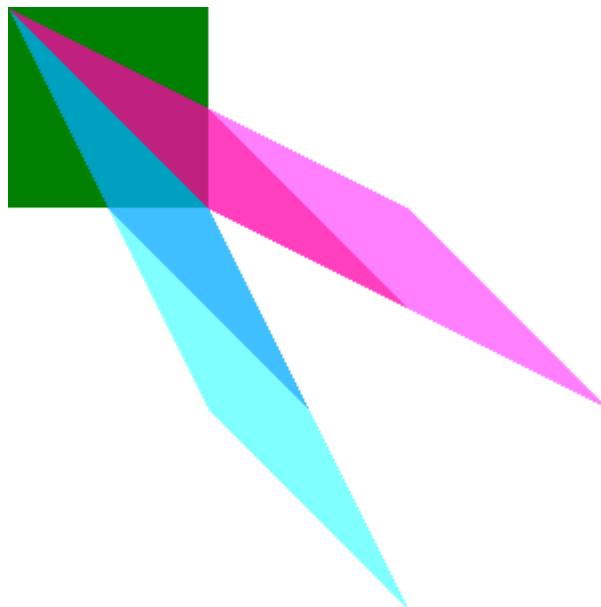
```

Graphics graph(pDC->m_hDC);
SolidBrush brush(Color::Green);
Rect rect(0, 0, 100, 100);
graph.FillRectangle(&SolidBrush(Color::Green), rect);
Matrix M;
M.Shear(1, 0.5f);
graph.SetTransform(&M);
graph.FillRectangle(&SolidBrush(Color(128, 255, 0, 0)), rect);
M.Reset();
M.Shear(0.5f, 1);
graph.SetTransform(&M);
graph.FillRectangle(&SolidBrush(Color(128, 0, 0, 255)), rect);
M.Reset();
M.Shear(2, 1);
graph.SetTransform(&M);
graph.FillRectangle(&SolidBrush(Color(128, 255, 0, 255)), rect);
M.Reset();
M.Shear(1, 2);
graph.SetTransform(&M);

```

```
graph.FillRectangle(&SolidBrush(Color(128, 0, 255, 255)), rect);
```

输出结果为：



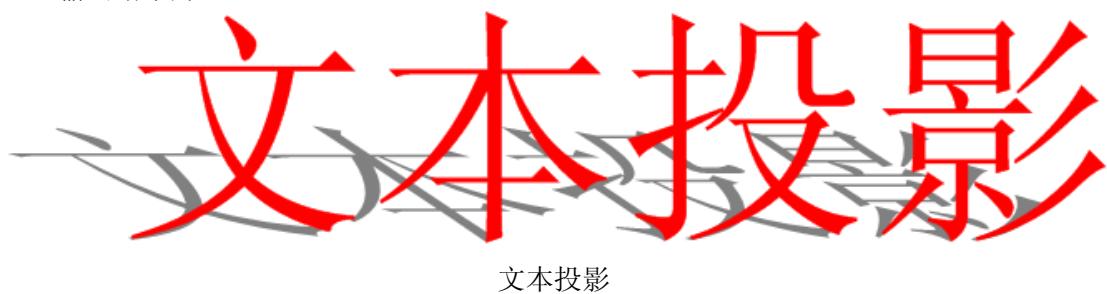
正方形（绿色）的剪切

红色-剪切(1, 0.5)、蓝色-剪切(0.5, 1)、品红-剪切(2, 1)、青色-剪切(1, 2)

又例如：（文本投影）

```
Graphics graph(pDC->m_hDC);
SolidBrush grayBrush(Color::Gray), redBrush(Color::Red);
Font font(L"宋体", 150, FontStyleRegular, UnitPixel);
Matrix M;
M.Shear(1.5f, 0.0f);
M.Scale(0.97f, 0.5f);
graph.SetTransform(&M);
graph.DrawString(L"文本投影", -1, &font, PointF(-168, 142), &grayBrush);
graph.ResetTransform();
graph.DrawString(L"文本投影", -1, &font, PointF(50, 0), &redBrush);
```

输出结果为：



(4) 变换点组

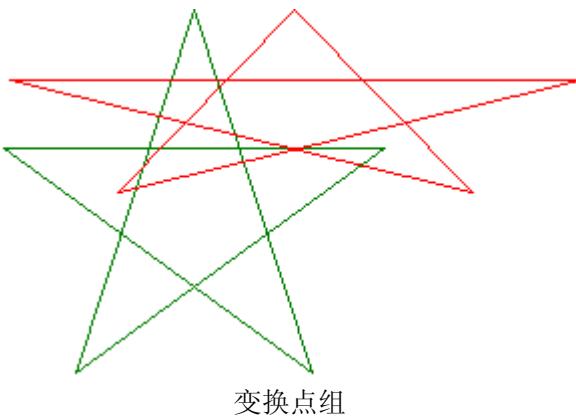
函数 TransformPoints 和 TransformVectors 是等价的，前者将坐标值对视为平面上的点，后者则视其为二维向量。两个函数都是用矩阵对一组点中的每个点进行坐标变换，得到新的点：

$$pts'[i] = pts[i] * M = \begin{pmatrix} pts[i].X & pts[i].Y & 1 \end{pmatrix} \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \\ dx & dy \end{pmatrix}$$

例如：

```
Graphics graph(pDC->m_hDC);
int n = 5;
Point pts[] = {Point(100, 0), Point(159, 181), Point(5, 69), Point(195, 69), Point(41, 181)};
graph.DrawPolygon(&Pen(Color::Green), pts, n);
Matrix M;
M.Scale(1.5f, 0.5f);
M.TransformPoints(pts, n);
graph.DrawPolygon(&Pen(Color::Red), pts, n);
```

输出结果为：



(5) 旋转变换

下面再通过两个例子，进一步介绍旋转矩阵变换在图形绘制中的作用。矩形类中有两个用于旋转的成员函数：

```
Status Rotate(REAL angle, MatrixOrder order = MatrixOrderPrepend); // 旋转
Status RotateAt(REAL angle, const PointF &center, // 相对于指定点旋转
    MatrixOrder order = MatrixOrderPrepend);
```

第一个函数 Rotate（相对于坐标原点[缺省为左上角]旋转）与图形类的 RotateTransform 成员函数等价，第二个函数 RotateAt（相对于指定点 center 旋转）则等价于图形类的两个成员函数 TranslateTransform 和 RotateTransform 的组合：

```
graph.TranslateTransform(center.X, center.Y); // 平移到旋转中心
graph.RotateTransform(angle, order); // 围绕中心旋转
```

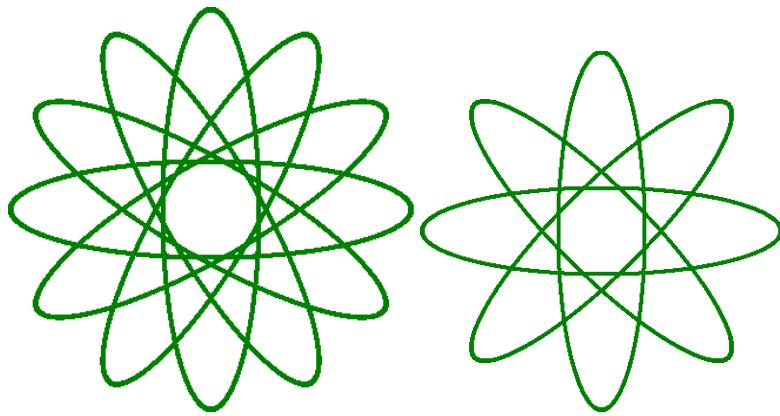
```

graph.TranslateTransform(-center.X, -center.Y); // 还原坐标原点
例如：（中心旋转椭圆）

Graphics graph(pDC->m_hDC);
Pen pen(Color::Green, 4);
pen.SetAlignment(PenAlignmentInset); // 画内接图形
CRect crect;
GetClientRect(&crect);
PointF cp(REAL(crect.CenterPoint().x), REAL(crect.CenterPoint().y));
INT w = min(crect.Width(), crect.Height()), h = w / 4;
Rect rect(INT(cp.X) - w / 2, INT(cp.Y) - h / 2, w, h);
Matrix M;
for (int i = 0; i < 180; i += 30/*45*/) {
    M.RotateAt(REAL(i), cp);
    graph.SetTransform(&M);
    graph.DrawEllipse(&pen, rect);
    M.Reset();
}

```

输出结果为：



中心旋转椭圆

又例如：（时钟）

可以利用 SDK 函数：

```
void GetLocalTime(LPSYSTEMTIME lpSystemTime);
```

来获得当前时间。其中的输入参数，为指向 SYSTEMTIME 结构的指针。

SYSTEMTIME 的定义为：

```

typedef struct _SYSTEMTIME {
    WORD wYear; // 年：1601~30827
    WORD wMonth; // 月：1~12
    WORD wDayOfWeek; // 星期几：0~6，其中 0 对应于星期日
    WORD wDay; // 日：1~31
    WORD wHour; // 时：0~23
    WORD wMinute; // 分：0~59
    WORD wSecond; // 秒：0~59
}
```

```

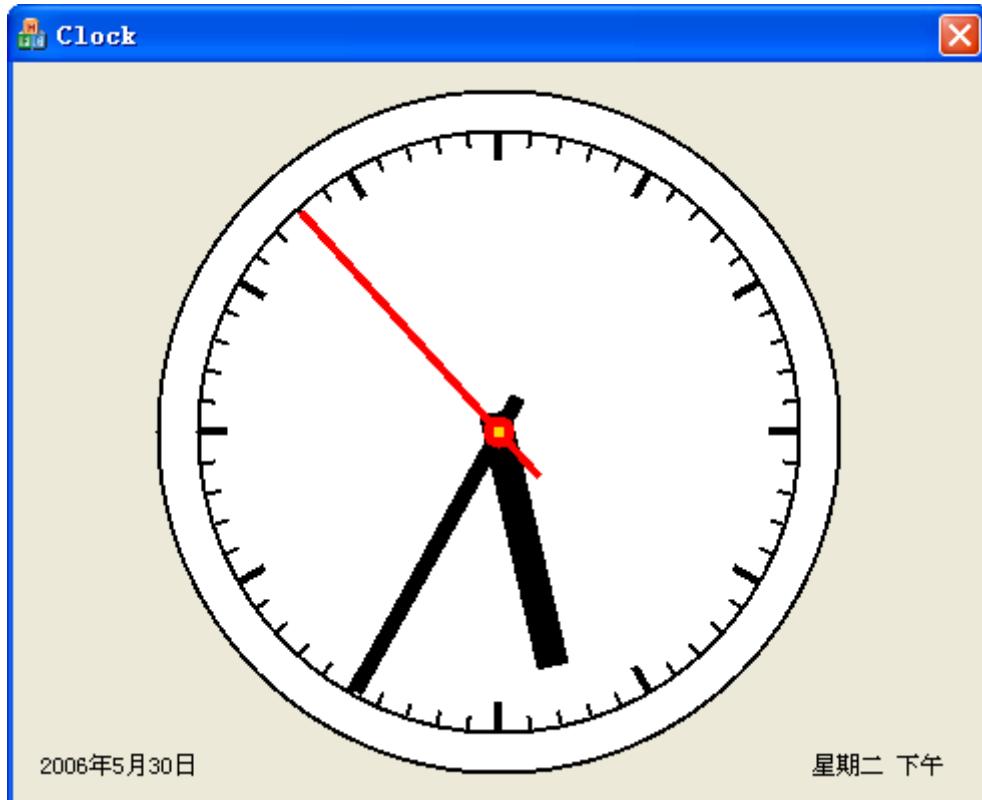
WORD wMilliseconds; // 毫秒: 0~999
} SYSTEMTIME;

// 画刻度盘
CPaintDC dc(this);
Graphics graph(dc.m_hDC);
//graph.SetSmoothingMode(SmoothingModeAntiAlias);
graph.TranslateTransform(cp.X, cp.Y);
graph.FillEllipse(pEraseBrush, *pOutCircleRect);
graph.DrawEllipse(&Pen(Color::Black, 2), *pInCircleRect);
graph.DrawEllipse(&Pen(Color::Black, 2), *pOutCircleRect);
graph.ResetTransform();
Matrix M;
for (int i = 0; i < 360; i += 6) {
    M.RotateAt(REAL(i), cp);
    graph.SetTransform(&M);
    graph.TranslateTransform(cp.X, cp.Y);
    if (i % 30) graph.DrawLine(p1mPen, r - dm, 0.0f, r, 0.0f);
    else graph.DrawLine(p5mPen, r - d5m, 0.0f, r, 0.0f);
    M.Reset();
}

```

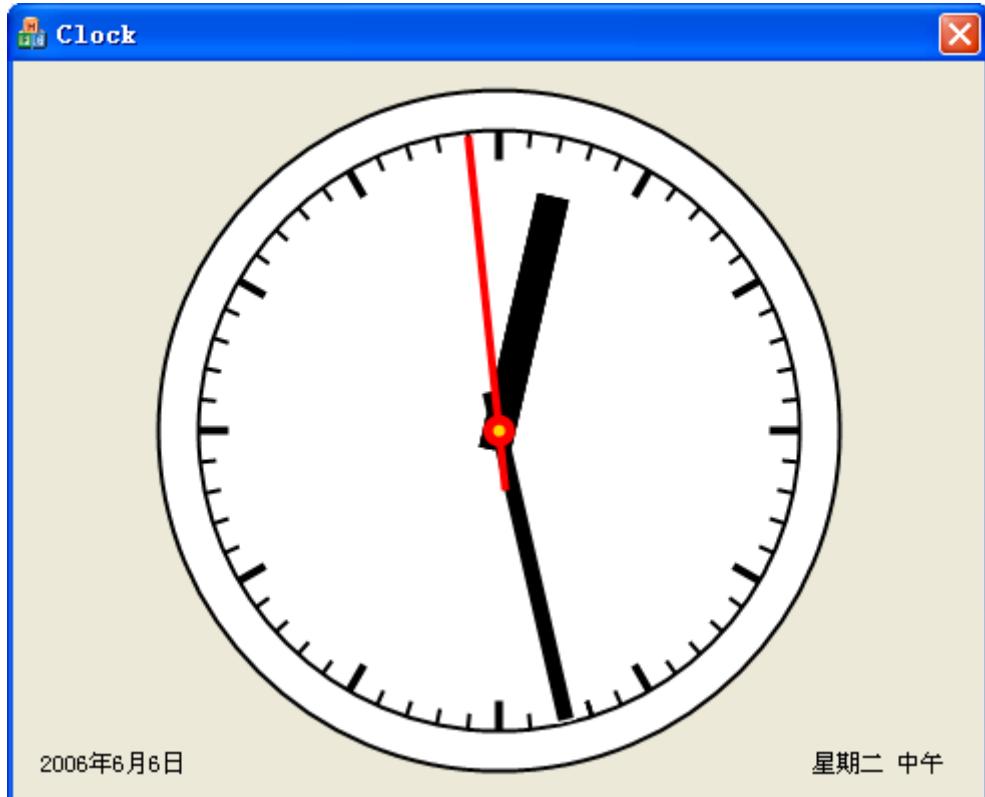
还要使用计时器及其消息响应，在该响应函数中画移动的指针（先画再用背景色擦除），可能还需要重画个别刻度。

程序的运行界面似：（完整程序留作作业）



时钟

加上平滑处理后的结果为：



平滑时钟

还可以每秒播放 Click.wav 波形文件资源，可以使用如下的 SDK 函数：

```
BOOL PlaySound(LPCSTR pszSound, HMODULE hmod, DWORD fdwSound);
```

例如：(其中 IDR_CLICK_WAVE 为已加入项目的波形声音资源的 ID)

```
PlaySound(MAKEINTRESOURCE(IDR_CLICK_WAVE),
```

```
    AfxGetInstanceHandle(), SND_RESOURCE);
```

另外，为了使包含 PlaySound 函数的程序能够编译通过，必须包含多媒体头文件：

```
#include <mmsystem.h>
```

并为项目添加多媒体库 winmm.lib (添加方法似 GDI+库)。

3) 路径变换

可以利用图形路径类 GraphicsPath 的成员函数：

```
Status Transform(const Matrix *matrix);
```

来对路径进行矩阵变换。

例如：(大小渐变文字)

```
GraphicsPath path; // 定义路径对象
path.AddString(L"大小渐变文字测试", -1, // 将文本串加入路径
    &FontFamily(L"隶书"), FontStyleRegular, 100, Point(0, 0), NULL);
RectF boundRect;
path.GetBounds(&boundRect); // 获取路径的界限矩形
```

```

Matrix M; // 定义矩阵对象（单位阵）
M.Translate(-(boundRect.X + boundRect.Width / 2), // 平移原点到文本路径的中心
            -(boundRect.Y + boundRect.Height / 2));
path.Transform(&M); // 更改路径的中心点
INT n = path.GetPointCount(); // 获取路径中的点数
PointF *points = new PointF[n]; // 动态创建点数组
path.GetPathPoints(points, n); // 获取路径的点数组
BYTE *types = new BYTE[n]; // 动态创建类型数组
path.GetPathTypes(types, n); // 获取路径类型数组（用于路径重构）
for (int i= 0; i < n; i++) // 根据路径点到中心的距离,按比例修改点的 y 值
    points[i].Y *= 2 * (boundRect.Width - abs(points[i].X)) / boundRect.Width;
GraphicsPath newPath(points, types, n); // 用新的路径点构造新路径
CRect crect;
GetClientRect(&crect);
Graphics graph(pDC->m_hDC); // 将坐标原点移到窗口中心:
graph.TranslateTransform(REAL(crect.Width() / 2), REAL(crect.Height() / 2));
graph.FillPath(&SolidBrush(Color::Green), &newPath); // 填充路径（绘制文本串）
输出结果为:

```

大小渐变文字测试

大小渐变文字

采用类似的方法，还可以让文本串沿圆周输出：（留作作业）



文本串的（半）圆形输出

4) 刷变换

纹理刷类 TextureBrush 和路径渐变刷类 PathGradientBrush，都支持基本的仿射变换——平移、旋转和伸缩：

```

Status TranslateTransform(REAL dx, REAL dy, MatrixOrder order = MatrixOrderPrepend);
Status RotateTransform(REAL angle, MatrixOrder order = MatrixOrderPrepend);

```

Status ScaleTransform(REAL sx, REAL sy, MatrixOrder order = MatrixOrderPrepend);
而且都可以设置复杂的矩阵变换:

Status SetTransform(const Matrix *matrix);

还都支持的矩阵变换:

Status MultiplyTransform(const Matrix* matrix, MatrixOrder order = MatrixOrderPrepend);
由于时间关系, 这里就不一一介绍了, 有兴趣的同学可以自己找资料看。

7. 图像

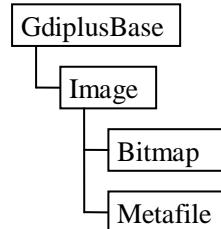
GDI+支持如下 9 种用于 Windows 的常见图像格式:

- **BMP**——BitMaP (位图), 扩展名为.BMP, 由 Microsoft 与 IBM 于 1980 年代中期为 Windows 和 PS/2 制订的图像格式, 一般不压缩。支持黑白、伪彩图、灰度图和真彩图, 每像素位数可为 1、4、8、16、24、32、64 等, 常用的是 24 位位图。
- **GIF**——Graphics Interchange Format (可交换图形格式), 扩展名为.GIF, 由 CompuServe 公司 1987 年制定, 采用无损的变长 LZW 压缩算法。只支持伪彩图 (最多 256 索引色), 宽高用双字节无符号整数表示 (最多 64K*64K 像素)。可存储多幅图片, 常用于简单的网络动画。压缩比较高, 使用广泛。
- **JPEG**——Joint Photographic Experts Group (联合图象专家组), 扩展名为.JPG, 是国际标准化组织 ISO 和 IEC 于 1991 年联合制定的静态图像压缩标准, 采用以 DCT 为主的有损压缩方法。支持灰度图和真彩图, 但是不支持伪彩图。压缩比高, 使用广泛。
- **Exif**——EXchangeable Image File Format (可交换图像文件格式), 扩展名为.Exif?, 由 JEIDA (Japan Electronic Industry Development Association 日本电子工业发展协会/日本电子情报技术产业协会) 于 1996 年 10 月制定。用于数码相机, 内含 JPEG 图像, 另包含拍摄日期、快门速度、曝光时间、照相机制造厂商和型号等相关信息。
- **PNG**——Portable Network Graphic Format (可移植网络图形格式, 读成 “ping”), 扩展名为.png, 由 W3C(World Wide Web Consortium 万维网协会)于 1996 年 10 月推出的一种标准图像格式, 2003 年成为 ISO 国际标准。PNG 采用与 GIF 一样的无损压缩方法, 但是除了伪彩图外, PNG 还支持多达 16 位深度的灰度图像和 48 位深度的彩色图像, 并且还可支持多达 16 位的 α 通道数据。
- **TIFF**——Tag Image File Format (标签图像文件格式), 扩展名为.tif, 由 Aldus 于 1986 年秋联合多家扫描仪制造商和软件公司共同开发, 支持黑白、索引色、灰度、真彩图, 可校正颜色和调色温, 支持多种压缩编码 (如 Huffman、LZW、RLE), 可存储多幅图片。常用于对质量要求高的专业图像的存储。
- **ICON**——icon(图标), 扩展名为.ico, 由 Microsoft 与 IBM 于 1980 年代中期为 Windows 和 PS/2 制订的图标图像格式。图像大小为 16*16、32*32 或 54*64。
- **WMF**——Windows MetaFile (视窗元文件), 扩展名为.WMF, 由 Microsoft 与 IBM 于 1980 年代中期为 Windows 和 PS/2 制订的图形文件格式, 用于保存 GDI 的绘图指令记录。
- **EMF**——Enhanced Windows MetaFile (增强型视窗元文件), 扩展名为.EMF, 是微软公司于 1993 年随 32 位操作系统 Windws NT 推出的一种改进的 WMF 格式, 用于 Win32。GDI+使用的是扩展 EMF 格式——EMF+。

GDI+的图像及其处理的功能十分强大, 可以用不同的格式加载、保存和操作图像。但

由于篇幅所限，本小节只介绍最基本的内容。

GDI+中有三个图像类，其中的 Image(图像)为基类，其他两个为它的派生类——Bitmap(位图)和 Metafile([图]元文件/矢量图形)。它们的类层次结构如下图所示：



图像类的层次结构

除此之外，还有大量与图像处理有关的 GDI+类，如 Effect 类及其 11 个派生类以及与图像数据和信息有关的 7 个独立类。由于时间关系，我们只准备介绍上面这三个主要的图像类及其基本操作。

1) 图像类 Image

Image 类是图像的基类，包含大量的通用图像操作的接口函数。

(1) 构造函数

Image 类有如下两个构造函数：

```
Image(const WCHAR *filename, BOOL useEmbeddedColorManagement = FALSE);  
Image(IStream *stream, BOOL useEmbeddedColorManagement = FALSE);
```

其中第一个较常用。它们的第二个参数都是用于颜色校正，一般取缺省值 FALSE 即可。

例如：

```
Image img(L"张东健.bmp");
```

或：

```
pImg = new Image(ar.GetFile()->GetFilePath()); // Image *pImg;
```

(2) 成员函数

Image 类的常用成员函数有：

```
UINT GetHeight(VOID); // 获取图像高度  
UINT GetWidth(VOID); // 获取图像宽度  
ImageType GetType(VOID); // 获取图像类型  
Status GetRawFormat(GUID *format); // 获取图像原始格式（图像文件类型）  
UINT GetFlags(VOID); // 获取图像标志  
Status GetLastStatus(VOID); // 获取最后的操作状态  
Image *Clone(VOID); // 克隆  
static Image *FromFile(const WCHAR *filename, // 由文件创建图像对象  
                      BOOL useEmbeddedColorManagement = FALSE);  
Status Save(const WCHAR *filename, const CLSID *clsidEncoder, // 保存图像到文件
```

```
const EncoderParameters *encoderParams = NULL);
```

(3) 获取类型与特征

- 获取图像对象种类

可以利用 `Image` 类的成员函数 `GetType` (获取类型) 来得到图像对象的种类:

```
ImageType GetType(VOID);
```

该函数返回的是 `ImageType` 枚举类型的值:

```
typedef enum {
    ImageTypeUnknown = 0, // 未知
    ImageTypeBitmap = 1, // 位图
    ImageTypeMetafile = 2 // 图元文件
} ImageType;
```

- 获取图像文件类型

可以利用 `Image` 类的成员函数 `GetRawFormat` (获取原始格式) 来得到图像文件的种类:

```
Status GetRawFormat(GUID *format);
```

该函数返回值的是如下 `GUID` (globally unique identifier 全局唯一标识符):

```
typedef struct _GUID {
    unsigned long Data1;
    unsigned short Data2;
    unsigned short Data3;
    unsigned char Data4[8];
} GUID;
```

结构常量的指针:

图像文件格式(11个)

ImageFormatBMP	BMP (BitMaP 位图)
ImageFormatEMF	EMF (Enhanced MetaFile 增强图元文件)
ImageFormatEXIF	Exif (Exchangeable Image File 可交换图像文件)
ImageFormatGIF	GIF (Graphics Interchange Format 图形交换格式)
ImageFormatIcon	Icon (图标)
ImageFormatJPEG	JPEG (Joint Photographic Experts Group 联合图象专家组)
ImageFormatMemoryBMP	从内存位图构造的图像
ImageFormatPNG	PNG (Portable Network Graphics 可移植网络图形)
ImageFormatTIFF	TIFF (Tagged Image File Format 标签图像文件格式)
ImageFormatUndefined	不能确定格式
ImageFormatWMF	WMF (Windows Metafile Format 视窗图元文件格式)

例如: JPEG 的 `GUID` 为 {B96B3CAE-0728-11D3-9D7B-0000F81EF32E}。

```
Image img(ar.GetFile()->GetFilePath());
GUID guid;
img.GetRawFormat(&guid);
if(guid == ImageFormatJPEG) MessageBox(L"The format is JPEG.");
.....
```

- 获取图像标志位

可以利用 Image 类的成员函数 GetFlags(获取标志位)来得到图像文件的各种特征标志:

```
UINT GetFlags(VOID);
```

该函数返回的是 ImageFlags 枚举类型的值:

```
typedef enum {
    ImageFlagsNone = 0, // 无格式信息
    ImageFlagsScalable = 0x0001, // 可缩放
    ImageFlagsHasAlpha = 0x0002, // 含 α 值
    ImageFlagsHasTranslucent = 0x0004, // 可半透明
    ImageFlagsPartiallyScalable = 0x0008, // 可部分缩放
    ImageFlagsColorSpaceRGB = 0x0010, // 颜色空间为 RGB
    ImageFlagsColorSpaceCMYK = 0x0020, // 颜色空间为 CMYK
    ImageFlagsColorSpaceGRAY = 0x0040, // 颜色空间为灰度
    ImageFlagsColorSpaceYCbCr = 0x0080, // 颜色空间为 YCbCr
    ImageFlagsColorSpaceYCCK = 0x0100, // 颜色空间为 YCCK
    ImageFlagsHasRealDPI = 0x1000, // 含有 DPI (dots per inch 每英寸点数) 信息
    ImageFlagsHasRealPixelSize = 0x2000, // 含有像素大小信息
    ImageFlagsReadOnly = 0x00010000, // 像素数据是只读的
    ImageFlagsCaching = 0x00020000 // 像素数据可被高速缓存
} ImageFlags;
```

2) 图像编解码与存储转换

保存和转换图像文件，需要使用编解码器的信息。

(1) 获取编解码器信息

可以用 GDI+的全局函数来获取系统提供的图像编解码器信息:

```
// 获取编码器总数和存储编码器信息所需的空间大小 (字节数):
Status GetImageEncodersSize(UINT *numEncoders, UINT *size);
// 获取编码器信息:
Status GetImageEncoders(UINT numEncoders, UINT size, ImageCodecInfo *encoders);
// 获取解码器总数和存储解码器信息所需的空间大小 (字节数):
Status GetImageDecodersSize(UINT *numDecoders, UINT *size);
// 获取解码器信息:
Status GetImageDecoders(UINT numDecoders, UINT size, ImageCodecInfo *decoders);
```

其中的编解码器信息类 ImageCodecInfo 中只含有一些数据成员:

数据成员	类型	描述
ClSID	CLSID	编解码器标识符
FormatID	GUID	文件格式标识符
CodecName	WCHAR *	编解码器名称串
DllName	WCHAR *	包含编解码器的 DLL 文件的路径名串

FormatDescription	WCHAR *	编解码器所使用的文件格式名串
FilenameExtension	WCHAR *	包含所有关联文件扩展名串（分号分隔）
MimeType	WCHAR *	包含编解码器的 MIME 类型串
Flags	DWORD	ImageCodecFlags 枚举的标志位组合
Version	DWORD	表示编解码器版本的整数
SigCount	DWORD	与编解码器关联的文件格式使用的签名数
SigSize	DWORD	表示每个签名的字节大小的整数
SigPattern	BYTE *	包含每个签名之模式的字节数组的指针
SigMask	BYTE *	包含每个签名之掩模的字节数组的指针

其中 MIME = Multipurpose Internet Mail Extension protocol, 多用途因特网邮件扩充协议

例如：(输出系统可用的编解码器信息)

```
void CImgProcView::OnCodecInfo()
{
    Graphics graph(GetDC()->m_hDC);
    RedrawWindow();
    SolidBrush brush(Color::Green);
    Font font(L"宋体", 12);

    UINT n, size;
    ImageCodecInfo *codecInfos;
    GetImageEncodersSize(&n, &size);
    codecInfos = (ImageCodecInfo *)malloc(size);
    GetImageEncoders(n, size, codecInfos);
    CString str = L"编解码器名称\t\t 格式描述\t 文件扩展名";
    graph.DrawString(str, str.GetLength(), &font, PointF(10, 10), &brush);
    for (UINT i = 0; i < n; i++) {
        str.Format(L"%s\t\t %s\t\t%s", codecInfos[i].CodecName,
                  codecInfos[i].FormatDescription, codecInfos[i].FilenameExtension);
        graph.DrawString(str, str.GetLength(), &font, PointF(10, 40 + 30 * REAL(i)), &brush);
    }
    free(codecInfos);
}
```

输出结果为：

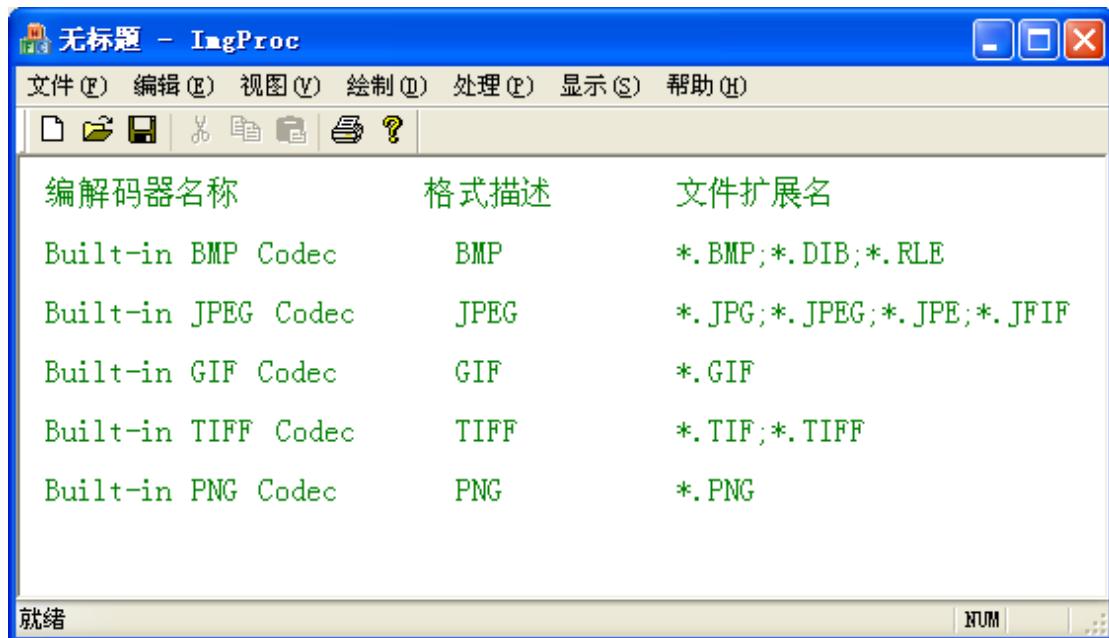


Image 类也有两个成员函数，可以用来获取编码器的参数列表：

```
UINT GetEncoderParameterListSize(const CLSID *clsidEncoder); // 返回参数表的大小（字节数）
Status GetEncoderParameterList(const CLSID *clsidEncoder, UINT size, // 获取参数表
EncoderParameters *buffer);
```

其中的 EncoderParameters 类，只包含两个数据成员：

```
UINT Count; // Number of EncoderParameter objects in the array.
EncoderParameter Parameter[1]; // Array of EncoderParameter objects.
```

其中的 EncoderParameter 类，包含如下 4 个数据成员：

```
GUID Guid; // Identifies the parameter category.
ULONG NumberOfValues; // Number of values in the array pointed to by the Value data member.
ULONG Type; // Identifies the data type of the parameter.
VOID *Value; // Pointer to an array of values.
```

(2) 图像的保存与转换

可以在对图像数据进行若干修改后，再保存到其他图像文件中（因为共享冲突，不能保存到原图像文件）。也可以将当前图像，用另一个指定的图像文件格式来保存。这些都可以用下面的 Image 类成员函数 Save 来完成：

```
Status Save(const WCHAR *filename, const CLSID *clsidEncoder,
const EncoderParameters *encoderParams = NULL);
Status Save(IStream *stream, const CLSID *clsidEncoder,
const EncoderParameters *encoderParams = NULL);
```

其中，需要类标识符 CLSID 的指针作为输入参数。为此，可以自己编写一个由格式描述符来获取类 ID 的函数。

例如：（我自己写的，供大家参考）

```
bool CImgProcDoc::GetEncoderClassID(const wchar_t *format, CLSID *pClsid)
{
    UINT n, size;
```

```

ImageCodecInfo *codecInfos;
GetImageEncodersSize(&n, &size);
codecInfos = (ImageCodecInfo *)malloc(size);
GetImageEncoders(n, size, codecInfos);
for (UINT i = 0; i < n; i++) {
    if (wcscmp(codecInfos[i].FormatDescription, format) == 0) {
        *pClsid = codecInfos[i].Clsid;
        free(codecInfos);
        return true;
    }
}
free(codecInfos);
return false;
}

```

下面是我自己写的一个保存图像为指定格式文件的信息响应函数(供大家参考):(其中的 fileName 为 CString 类变量,是在图像文件被装入时,得到的当前图像文件名。这里用作缺省的保存文件名)

```

void CImgProcDoc::OnFileSaveAs() {
    if (ok) {
        wchar_t filters[] = L"图像文件 (*.bmp;*.gif;*.jpg;*.png;*.tif;*.emf)|*.bmp;*.gif;*.jpg;\\
*.png;*.tif;*.emf|位图文件 (*.bmp)|*.bmp|图形交换格式文件 (*.gif)|*.gif|联合图象专家组\\
[JPEG]文件 (*.jpg)|*.jpg|可移植网络图形文件 (*.png)|*.png|标记图像文件格式[TIFF]\\
文件 (*.tif)|*.tif|增强型图元文件 (*.emf)|*.emf|所有文件 (*.*)|*.*||";
        CFileDialog fileDlg(FALSE, NULL, fileName, OFN_HIDEREADONLY, filters);
        if (fileDlg.DoModal() == IDOK) {
            CString extStr = fileDlg.GetFileExt();
            extStr.MakeUpper();
            if (extStr.Compare(L"JPG") == 0) extStr = L"JPEG";
            else if (extStr.Compare(L"TIF") == 0) extStr = L"TIFF";
            CLSID clsid;
            if (GetEncoderClassID(extStr, &clsid)) {
                pImg->Save(fileDlg.GetPathName(), &clsid, NULL);
                if (pImg->GetLastStatus() != Ok)
                    MessageBox(NULL, L"保存文件出错!", L"错误", MB_OK);
            }
            else MessageBox(NULL, L"找不到指定的编码器!", L"错误", MB_OK);
        }
    }
}

```

说明: 这里需要自己为文档类添加“另存为”菜单项 ID_FILE_SAVE_AS 的消息响应函数,还可以将“保存”菜单项的消息响应也定向于该函数。例如,可以在消息映射

ON_COMMAND(ID_FILE_SAVE_AS, &CImgProcDoc::**OnFileSaveAs**)
之后, 手工添加消息映射:

```
ON_COMMAND(ID_FILE_SAVE, &CImgProcDoc::OnFileSaveAs)
```

3) 位图类 Bitmap

在三个图像类中，Bitmap 类对应于点阵位图，Metafile 类对应于矢量图形，基类 Image 则对应于通用操作。

(1) 构造函数

Bitmap 类有 10 个构造函数，其中常用的有：

```
Bitmap(const WCHAR *filename, BOOL useIcm = FALSE);  
Bitmap(INT width, INT height, PixelFormat format = PixelFormat32bppARGB);  
Bitmap(INT width, INT height, Grpahics *target);
```

其中 PixelFormat 可取值为下列常量：

图像像素格式(14 个)

PixelFormat1bppIndexed	每像素 1 位，索引色
PixelFormat4bppIndexed	每像素 4 位，索引色
PixelFormat8bppIndexed	每像素 8 位，索引色
PixelFormat16bppARGB1555	每像素 16 位， α 分量 1 位、RGB 分量各 5 位
PixelFormat16bppGrayScale	每像素 16 位，灰度
PixelFormat16bppRGB555	每像素 16 位，RGB 分量各 5 位，另 1 位未用
PixelFormat16bppRGB565	每像素 16 位，RB 分量各 5 位、G 分量 6 位
PixelFormat24bppRGB	每像素 24 位，RGB 分量各 8 位
PixelFormat32bppARGB	每像素 32 位， α RGB 分量各 8 位
PixelFormat32bppPARGB	每像素 32 位， α RGB 分量各 8 位，RGB 分量预乘 α 分量
PixelFormat32bppRGB	每像素 24 位，RGB 分量各 8 位，另 8 位未用
PixelFormat48bppRGB	每像素 48 位，RGB 分量各 16 位
PixelFormat64bppARGB	每像素 64 位， α RGB 分量各 16 位
PixelFormat64bppPARGB	每像素 64 位， α RGB 分量各 16 位，RGB 分量预乘 α 分量

(2) 成员函数

Bitmap 类的专有成员函数有：

```
// 克隆  
Bitmap *Clone(INT x, INT y, INT width, INT height, PixelFormat format);  
Bitmap *Clone(const Rect &rect, PixelFormat format);  
Bitmap *Clone(REAL x, REAL y, REAL width, REAL height, PixelFormat format);  
Bitmap *Clone(const RectF &rect, PixelFormat format);  
  
// 创建  
static Bitmap *FromBITMAPINFO(const BITMAPINFO *gdiBitmapInfo, VOID *gdiBitmapData);  
static Bitmap *FromDirectDrawSurface7(IDirectDrawSurface7* surface);  
static Bitmap *FromFile(const WCHAR *filename, BOOL useEmbeddedColorManagement = FALSE);
```

```

static Bitmap *FromHBITMAP(HBITMAP hbm, HPALETTE hpal);
static Bitmap *FromHICON(HICON hicon);
static Bitmap *FromResource(HINSTANCE hInstance, const WCHAR *bitmapName);
static Bitmap *FromStream(IStream *stream, BOOL useEmbeddedColorManagement);
// 获取
Status GetHBITMAP(const Color &colorBackground, HBITMAP *hbmReturn);
Status GetHICON(HICON *hicon);
// 像素
Status GetPixel(INT x, INT y, Color *color);
Status SetPixel(INT x, INT y, const Color &color);
// 设置
Status SetResolution(REAL xdpi, REAL ydpi);
Status LockBits(const Rect *rect, UINT flags, PixelFormat format, BitmapData *lockedBitmapData);
Status UnlockBits(BitmapData *lockedBitmapData);

```

4) 基本操作

(1) 绘制图像

图形类 Graphics 有 18 个同名的重载成员函数用于绘制图像, 其中常用的有如下 4 个(它们都有对应的浮点数版):

```

Status DrawImage(Image *image, INT x, INT y); // 不缩放 (坐标对)
Status DrawImage(Image *image, const Point &point); // 不缩放 (点)
Status DrawImage(Image *image, INT x, INT y, INT width, INT height); // 可缩放 (坐标对与宽高)
Status DrawImage(Image *image, const Rect &rect); // 可缩放 (矩形)

```

例如:

```

Graphics graph(pDC->m_hDC);
Image img(L"张东健.bmp");
graph.DrawImage(&img, 0, 0);

```

或:

```

Graphics graph(pDC->m_hDC);
Image img(L"金泰熙.bmp");
CRect rect;
GetClientRect(&rect);
graph.DrawImage(&img, 0, 0, rect.Width(), rect.Height());

```

(2) 缩放

图形类 Graphics 的图像绘制函数:

```

Status DrawImage(Image *image, INT x, INT y, INT width, INT height);
Status DrawImage(Image *image, const Rect &rect);
Status DrawImage(Image *image, REAL x, REAL y, REAL width, REAL height);
Status DrawImage(Image *image, const RectF &rect);

```

可以缩放图像。(与 GDI 的 StretchBlt 函数的功能类似)

例如：(按客户区大小缩放)

```
Graphics graph(pDC->m_hDC);
Image img(L"金泰熙.bmp");
CRect rect;
GetClientRect(&rect);
graph.DrawImage(&img, 0, 0, rect.Width(), rect.Height());
```

输出结果为：



按客户区大小缩放

又例如：(按指定[实数]倍数 zoomMultiple 缩放)

```
Graphics graph(pDC->m_hDC);
Image img(L"金泰熙.bmp");
graph.DrawImage(&img, 0.0f, 0.0f, zoomMultiple * img.GetWidth(),
               zoomMultiple * img.GetHeight());
```

输出结果为：



按指定[实数]倍数缩放 (zoomMultiple = 1.5)

还可以使用 Graphics 类的设置插值模式成员函数：

 Status SetInterpolationMode(InterpolationMode interpolationMode);
来控制缩放的质量。其中的输入参数为枚举类型 InterpolationMode：

```
typedef enum {  
    InterpolationModeInvalid = QualityModeInvalid, // 无效插值  
    InterpolationModeDefault = QualityModeDefault, // 缺省插值  
    InterpolationModeLowQuality = QualityModeLow, // 低质插值  
    InterpolationModeHighQuality = QualityModeHigh, // 高质插值  
    InterpolationModeBilinear = QualityModeHigh + 1, // 双线性插值  
    InterpolationModeBicubic = QualityModeHigh + 2, // 双三次插值  
    InterpolationModeNearestNeighbor = QualityModeHigh + 3, // 最邻近插值  
    InterpolationModeHighQualityBilinear = QualityModeHigh + 4, // 高质双线性插值  
    InterpolationModeHighQualityBicubic = QualityModeHigh + 5 // 高质双三次插值  
} InterpolationMode;
```

例如：

```
Graphics graph(pDC->m_hDC);  
Image img(L"张东健.bmp"));
```

```

//graph.SetInterpolationMode(InterpolationModeNearestNeighbor);
//graph.SetInterpolationMode(InterpolationModeBilinear);
graph.SetInterpolationMode(InterpolationModeHighQualityBilinear);
graph.DrawImage(&img, 0.0f, 0.0f, 3.23f *img.GetWidth(), 3.23f * img.GetHeight());

```

输出结果为：



(3) 剪裁与局部缩放

可以使用图形类 Graphics 的图像绘制函数：

```

Status DrawImage(Image *image, INT x, INT y, INT srcx, INT srcy,
                 INT srcwidth, INT srcheight, Unit srcUnit); // 不缩放（整数版）
Status DrawImage(Image *image, REAL x, REAL y, REAL srcx, REAL srcy,
                 REAL srcwidth, REAL srcheight, Unit srcUnit); // 不缩放（实数版）
Status DrawImage(Image* image, const RectF& destRect, REAL srcx, REAL srcy,
                 REAL srcwidth, REAL srcheight, Unit srcUnit, const ImageAttributes*
                 imageAttributes = NULL, DrawImageAbort callback = NULL, VOID*
                 callbackData = NULL); // 可缩放（实数版）
Status DrawImage(Image *image, RectF &destRect, RectF &sourceRect, // 该函数库中无
                 Unit srcUnit, ImageAttributes *imageAttributes = NULL); // 可缩放（实数版）

```

来绘制图像的指定区域，即将图像剪裁后再输出。

例如：(其中，cx、cy、cw、ch 为剪裁区域的左上角坐标和宽高，zoomMultiple 为缩放倍数)

```

Graphics graph(pDC->m_hDC);
Image img(L"张东健.bmp");
graph.DrawImage(&img, 0, 0, cx, cy, cw, ch, UnitPixel); // 不缩放
CRect rect;
GetClientRect(&rect);
graph.DrawImage(&img, RectF(0, 0, REAL(rect.Width()), REAL(rect.Height())),
               REAL(cx), REAL(cy), REAL(cw), REAL(ch), UnitPixel); // 客户区缩放

```

```
graph.DrawImage(&img, RectF(0, 0, zoomMultiple * cw, zoomMultiple * ch),  
    REAL(cx), REAL(cy), REAL(cw), REAL(ch), UnitPixel); // 倍数缩放
```

输出结果如：



剪裁及其缩放

左上角为原始图像，左中为剪裁区域，左下为剪裁后输出的图像
右上与下中为剪裁图随客户区缩放，右下为倍数放大（10 倍）

(4) 旋转与翻转

可以使用 Image 类的旋转翻转成员函数

```
Status RotateFlip(RotateFlipType rotateFlipType);
```

来对图像进行旋转和翻转。其中的输入参数的取值为枚举类型 RotateFlipType 常量：

```
typedef enum {  
    RotateNoneFlipNone = 0, // 不旋转不翻转  
    Rotate90FlipNone = 1, // 旋转 90 度不翻转  
    Rotate180FlipNone = 2, // 旋转 180 度不翻转  
    Rotate270FlipNone = 3, // 旋转 270 度不翻转  
    RotateNoneFlipX = 4, // 不旋转 x 向翻转  
    Rotate90FlipX = 5, // 旋转 90 度 x 向翻转  
    Rotate180FlipX = 6, // 旋转 180 度 x 向翻转  
    Rotate270FlipX = 7, // 旋转 270 度 x 向翻转  
    RotateNoneFlipY = Rotate180FlipX, // 不旋转 y 向翻转  
    RotateNoneFlipY = Rotate270FlipX, // 旋转 90 度 y 向翻转  
    Rotate180FlipY = RotateNoneFlipX, // 旋转 180 度 y 向翻转  
    Rotate270FlipY = Rotate90FlipX, // 旋转 270 度 y 向翻转  
    RotateNoneFlipXY = Rotate180FlipNone, // 不旋转 xy 向翻转  
    Rotate90FlipXY = Rotate270FlipNone, // 旋转 90 度 xy 向翻转  
    Rotate180FlipXY = RotateNoneFlipNone, // 旋转 180 度 xy 向翻转
```

```
    Rotate270FlipXY = Rotate90FlipNone // 旋转 270 度 xy 向翻转  
} RotateFlipType;
```

下面是各个常量的含义：



RotateNoneFlipNone
Rotate180FlipXY

Rotate90FlipNone
Rotate270FlipXY

Rotate180FlipNone
RotateNoneFlipXY

Rotate270FlipNone
Rotate90FlipXY



RotateNoneFlipX
Rotate180FlipY

Rotate90FlipX
Rotate270FlipY

Rotate180FlipX
RotateNoneFlipY

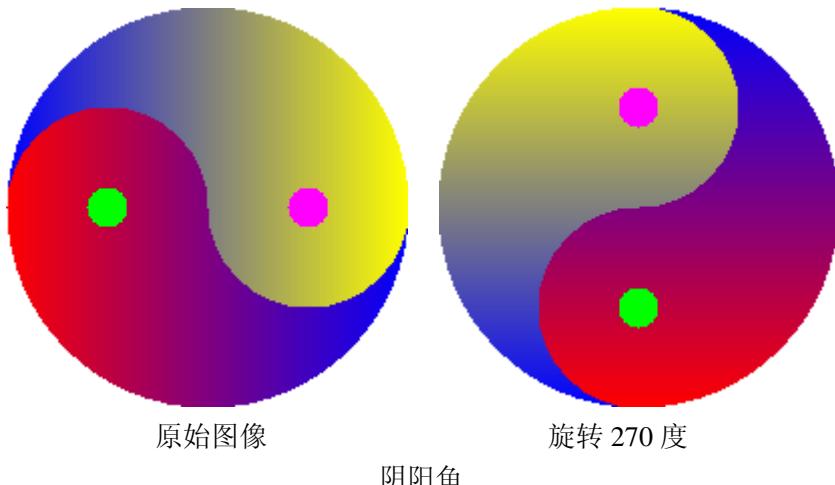
Rotate270FlipX
RotateNoneFlipY

旋转和翻转

例如：（旋转阴阳鱼）

```
Graphics graph(pDC->m_hDC);  
Image img(L"yyy.emf");  
img.RotateFlip(RotateFlipType::Rotate270FlipNone);  
graph.DrawImage(&img, 0, 0);
```

输出结果为：



注意，每次对图像的旋转和翻转，都是在上次已经进行过的旋转和翻转的结果图像上，再实施指定的旋转角度的翻转方向（即是相对于当前图像进行接连操作，而不是相对于原始图像来操作的）。

实施旋转（90 度或 270 度的）操作后，图像的长和宽被对换，需要重新获取图像的长和宽。

（5）仿射变形

可以利用 Graphics 类的成员函数：

```
Status DrawImage(Image *image, const Point *destPoints, INT count);
```

```
Status DrawImage(Image *image, const PointF *destPoints, INT count);
```

将矩形图像绘制到一个平行四边形中，可用于立体投影面图的绘制。

其中，count 必须为 3，destPoints 中含目标区域——平行四边形的 3 个顶点（依次为）：左上角、右上角和左下角。

例如：（立方体贴图）

```
Graphics graph(GetDC()->m_hDC);
Image img1(L"刘亦菲.bmp"), img2(L"张含韵.bmp"), img3(L"金泰熙.bmp");
int d = 10, dx = (2 * img1.GetWidth()) / 3, dy = img3.GetHeight() / 2;
Point p1(d, d + dy), p2(p1.X + img1.GetWidth(), p1.Y), p3(p2.X + dx, d),
      p4(p2.X, p2.Y + img1.GetHeight()), p5(p1.X + dx, d);
graph.DrawImage(&img1, p1);
Point pts1[] = {p2, p3, p4}, pts2[] = {p5, p3, p1};
graph.DrawImage(&img2, pts1, 3);
graph.DrawImage(&img3, pts2, 3);
cube = true;
```

输出结果为：



立方体贴图

(6) 多帧图像与动画

有些图像文件（如 GIF 和 TIFF）中，可包含多幅图像。下面这些 Image 类的成员函数可以用于多帧图像的信息获取和当前图像帧的设置：

```
UINT GetFrameDimensionsCount(VOID); // 获取图像帧维数（不同的图像类型数）
Status GetFrameDimensionsList(GUID *dimensionIDs, UINT count); // 获取帧维列表
UINT GetFrameCount(const GUID *dimensionID); // 获取图像帧数
Status SelectActiveFrame(const GUID *dimensionID, UINT frameIndex); // 选择当前活动帧

例如：（类变量：UINT cf = 0, fn, dn;）

// 在动画菜单项的消息响应函数中：
dn = pImg->GetFrameDimensionsCount();
guids = new GUID[dn];
pImg->GetFrameDimensionsList(guids, dn);
fn = pImg->GetFrameCount(&guids[0]);
if (fn < 2) { // res\Images 资源目录中的 globe.gif 文件，是一个地球 GIF 动画文件
    MessageBox(L"图像文件中未含多帧图片！");
    return;
}
SetTimer(1, 100, NULL);

// 在计时器的消息响应函数中：
pImg->SelectActiveFrame(guids, cf);
Graphics graph(GetDC()->m_hDC);
graph.DrawImage(pImg, 0, 0);
cf++;
cf %= fn;
```

还可以用 Image 类的成员函数 SaveAdd 来保存多帧图像：（需要先调用 Save 后再调用）

```
Status SaveAdd(Image *newImage, const EncoderParameters *encoderParams);
Status SaveAdd(const EncoderParameters* encoderParams);
```

例如：（创建多帧 TIFF 文件[GIF 文件还不太听话]）

```
EncoderParameters encoderParameters; // 编码参数对象数组
ULONG parameterValue; // 编码值
encoderParameters.Count = 1; // 只有一个参数对象
encoderParameters.Parameter[0].Guid = EncoderSaveFlag; // 保存标志
// 参数为长整数类型
encoderParameters.Parameter[0].Type = EncoderParameterValueTypeLong;
encoderParameters.Parameter[0].NumberOfValues = 1; // 只有一个参数
encoderParameters.Parameter[0].Value = &parameterValue; // 参数值的地址
Bitmap mimg(L"Duke\\T1.BMP"); // 创建多帧图像对象，并装入第一个图片
CLSID clsid; // 类 ID
if (GetDocument()->GetEncoderClassID(L"TIFF", &clsid)) { // 获取类 ID
    parameterValue = EncoderValueMultiFrame; // 设置多帧格式
    mimg.Save(L"Duke.tif", &clsid, &encoderParameters); // 保存第一帧图片
```

```

if (mimg.GetLastStatus() != Ok) { // 出错显示提示信息并返回
    MessageBox(L"保存文件出错！");
    return;
}
}else {
    MessageBox(L"找不到指定的编码器！");
    return;
}
wchar_t fnStr[256];
for (int i = 2; i <= 10; i++) {
    parameterValue = EncoderValueFrameDimensionPage; // 设置多帧子图片格式
    swprintf_s(fnStr, 256, L"Duke\%T%d.BMP", i); // 设置文件名
    mimg.SaveAdd(&Image(fnStr), &encoderParameters); // 添加新图片
    if (mimg.GetLastStatus() != Ok) { // 出错显示提示信息并返回
        MessageBox(L"添加文件出错！");
        return;
    }
}
}

```

5) 调整色彩

GDI+中关于调整图像颜色的功能十分强大，主要封装在图像属性类 `ImageAttributes` 中。该类只有一个缺省构造函数：

`ImageAttributes(VOID);`

但是却有很多其他可用于颜色调整的成员函数。例如，可以用 `ImageAttributes` 类的重置成员函数：

`Status Reset(ColorAdjustType type = ColorAdjustTypeDefault);`

来取消已经进行了的各种属性设置。

我们这里只介绍其中最简单的几种功能。

(1) γ 曲线校正

各种图像设备的光电转换特性存在差异，使得在不同设备上再现图像时，必须进行亮度和对比度上的调整，这在专业上是通过 γ (Gamma) 曲线校正来完成。

当 γ 校正值 >1 时，图像的高光部分被压缩，而暗调部分被扩展；当 γ 校正值 <1 时，图像的高光部分被扩展，而暗调部分被压缩；当 γ 校正值 $=1$ 时，图像的亮度信息保持不变。 γ 校正一般用于平滑地扩展暗调的细节。

在 GDI+中，可以先调用 `ImageAttributes` 类的设置 γ 值成员函数：

`Status SetGamma(REAL gamma, ColorAdjustType type = ColorAdjustTypeDefault);`

来设置图像的 γ 值。其中的颜色调整类型参数 `type` 取枚举类型的常量值：

```

typedef enum {
    ColorAdjustTypeDefault = 0, // 缺省
    ColorAdjustTypeBitmap = 1, // 位图
}

```

```

ColorAdjustTypeBrush = 2, // 刷
ColorAdjustTypePen = 3, // 笔
ColorAdjustTypeText = 4, // 图元文件中的文本
ColorAdjustTypeCount = 5, // 内部调整类型计数 (可用于循环)
ColorAdjustTypeAny = 6 // 保留
} ColorAdjustType;

```

然后再使用 Graphics 类的几个带图像属性类对象参数的绘制图像成员函数

```

Status DrawImage(Image *image, RectF &destRect, RectF &sourceRect, Unit srcUnit,
ImageAttributes *imageAttributes); // 该函数库中无 (帮助文档中有)

Status DrawImage(Image *image, const Rect &destRect, INT srcx, INT srcy, INT
srcwidth, INT srcheight, Unit srcUnit = UnitPixel, ImageAttributes
*imageAttributes = NULL, DrawImageAbort callback = NULL, VOID
*callbackData = NULL);

Status DrawImage(Image *image, const Point *destPoints, INT count, INT srcx, INT
srcy, INT srcwidth, INT srcheight, Unit srcUnit = UnitPixel, ImageAttributes
*imageAttributes = NULL, DrawImageAbort callback = NULL, VOID
*callbackData = NULL);

```

来绘制经过 γ 校正后图像。

还可以用下面的 ImageAttributes 类的成员函数

```

Status ClearGamma(ColorAdjustType type = ColorAdjustTypeDefault);

```

来清除 γ 校正的设置。

例如：

```

Graphics graph(GetDC()->m_hDC);
Image img(L"金泰熙.bmp");
int w = img.GetWidth(), h = img.GetHeight();
ImageAttributes imgAttr;
imgAttr.SetGamma(2); // 0.5f
graph.DrawImage(&img, Rect(0, 0, w, h), 0, 0, w, h, UnitPixel, &imgAttr);

```

输出结果为：



原图

$\gamma = 2$

$\gamma = 0.5$

γ 校正

(2) 颜色通道

可以用 ImageAttributes 类的设置输出通道的成员函数:

```
Status SetOutputChannel(ColorChannelFlags channelFlags, ColorAdjustType type = ColorAdjustTypeDefault);
```

来设置图像的颜色输出通道。其中，颜色通道标志参数取枚举类型：

```
typedef enum {
    ColorChannelFlagsC = 0, // 青色 (cyan) 通道
    ColorChannelFlagsM = 1, // 品红 (magenta) 通道
    ColorChannelFlagsY = 2, // 黄色 (yellow) 通道
    ColorChannelFlagsK = 3, // 黑色 (black) 通道
    ColorChannelFlagsLast = 4 // 最后值标志 (可用于循环)
} ColorChannelFlags;
```

其中 CMY 为色料三原色，主要用于打印和印刷部门。

还可以用下面的 ImageAttributes 类的成员函数

```
Status ClearOutputChannel(ColorAdjustType type = ColorAdjustTypeDefault);
```

来清除颜色通道的设置。

例如：(ColorChannelFlags colChannel;的值由用户设置)

```
Graphics graph(GetDC()->m_hDC);
Image img(L"张含韵.bmp");
int w = img.GetWidth(), h = img.GetHeight();
ImageAttributes imgAttr;
imgAttr.SetOutputChannel(colChannel);
graph.DrawImage(&img, Rect(0, 0, w, h), 0, 0, w, h, UnitPixel, &imgAttr);
```

输出结果为：



原图

青色 C

品红 M
黄色通道

黄色 Y

黑色 K

(3) 颜色阈值

颜色的阈值 (threshold) (取值：0 ~ 1) 一般用于图像的简单锐化。具体做法是，将图像中每个像素的各个颜色分量值，与最大值 (255) 和阈值的乘积相比较，大于该乘积的分

量的值被设为 255，小于和等于该乘积的分量的值被设为 0。结果，整幅图只剩下 $2 \times 2 \times 2 = 8$ 种颜色。

可以用 ImageAttributes 类的设置阈值成员函数：

Status SetThreshold(REAL threshold, ColorAdjustType type = ColorAdjustTypeDefault);
来设置图像的颜色阈值。

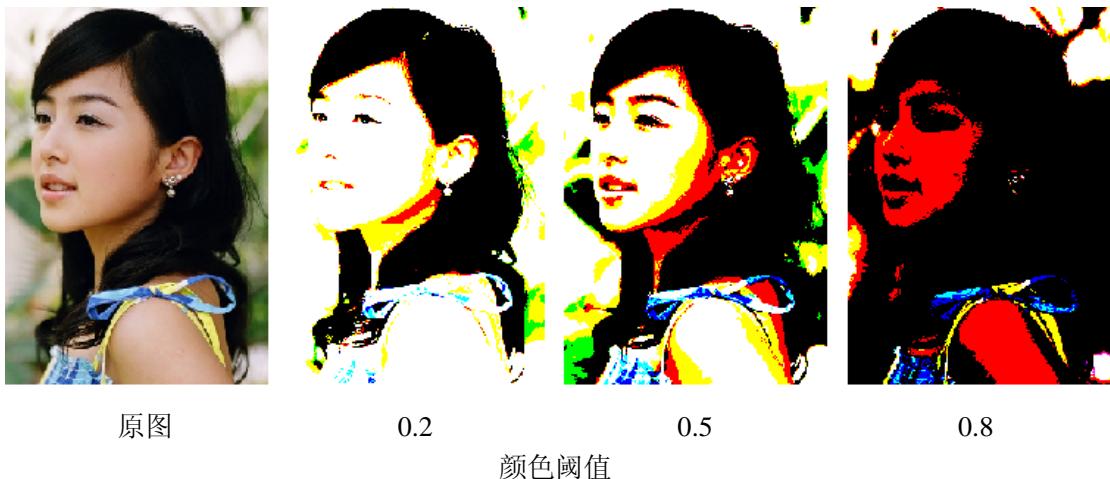
也可以用下面的 ImageAttributes 类的成员函数

Status ClearThreshold(ColorAdjustType type = ColorAdjustTypeDefault);
来清除颜色阈值的设置。

例如：

```
Graphics graph(GetDC()->m_hDC);
Image img(L"张含韵.bmp");
int w = img.GetWidth(), h = img.GetHeight();
ImageAttributes imgAttr;
imgAttr.SetThreshold(0.5f); // 0.2f ~ 0.8f
graph.DrawImage(&img, Rect(0, 0, w, h), 0, 0, w, h, UnitPixel, &imgAttr);
```

输出结果为：



还可以利用 ImageAttributes 类的成员函数，来设置关键色范围、设置画刷的颜色映射表、设置彩色和灰度校正矩阵、设置输出通道颜色配置文件、设置颜色映射表等等。由于时间关系，这里就不再介绍了。有兴趣的同学，可以自己去看查考书和各种资料。

6) 特技处理

下面介绍几种图像的特技处理方法，包括灰化、负片、木刻、雕刻和浮雕等。

(1) 灰化

灰化 (gray) 指将彩色图像变成灰度图，标准的处理办法，是根据人眼对不同色彩的强度感知不同，采取加权平均的方法：

$$r = g = b = L = 0.299R + 0.587G + 0.114B$$

也可以简单地用算术平均来进行灰化：

$$r = g = b = L = (R + G + B) / 3$$

例如：(其中 pImg 为指向原图像对象的指针， w 和 h 为其宽和高。下同)

```
Color col;  
BYTE val;  
pGrayImg = new Bitmap(w, h);  
for (int i = 0; i < w; i++) {  
    for (int j = 0; j < h; j++) {  
        pImg->GetPixel(i, j, &col);  
        val = BYTE(0.299f * col.GetR() + 0.587f * col.GetG() + 0.114 * col.GetB());  
        //val = BYTE((col.GetR() + col.GetG() + col.GetB()) / 3);  
        col.SetValue(col.MakeARGB(255, val, val, val));  
        pGrayImg->SetPixel(i, j, col);  
    }  
}
```

输出结果如：



(2) 负片

负片 (negative 底片) 的制作很简单，只需将原图像每个像素的三个颜色分量 v 都取反 ($255 - v$) 即可。

例如：

```
Color col;  
BYTE r, g, b;  
pNegativeImg = new Bitmap(w, h);  
for (int i = 0; i < w; i++) {  
    for (int j = 0; j < h; j++) {  
        pImg->GetPixel(i, j, &col);  
        r = 255 - col.GetR();  
        g = 255 - col.GetG();
```

```

        b = 255 - col.GetB();
        col.SetValue(col.MakeARGB(255, r, g, b));
        pNegativeImg->SetPixel(i, j, col);
    }
}

```

输出结果如：



原图像



负片
负片效果

(3) 木刻

这里的木刻（woodcut）是指，将原来彩色图像，变成黑白二值图像。以像素各颜色分量的平均值（即灰度值），是否达到或超过亮度的中间值（128）来区分。亮的像素被置为白色（ $r = g = b = 255$ ）、暗的像素被置为黑色（ $r = g = b = 0$ ）。

与灰化方法一样，也有两种计算平均值的方法：

加权平均法：

$$\text{avg} = 0.299R + 0.587G + 0.114B$$

算术平均法：

$$\text{avg} = (R + G + B) / 3$$

例如：

```

Color col;
int avg;
BYTE val;
pWoodcutImg = new Bitmap(w, h);
for (int i = 0; i < w; i++) {
    for (int j = 0; j < h; j++) {
        pImg->GetPixel(i, j, &col);
        avg = int(0.299f * col.GetR() + 0.587f * col.GetG() + 0.114 * col.GetB());
        //avg = int((col.GetR() + col.GetG() + col.GetB()) / 3);
        if (avg >= 128) val = 255; else val = 0;
        col.SetValue(col.MakeARGB(255, val, val, val));
    }
}

```

```

    pWoodcutImg->SetPixel(i, j, col);
}
}

```

输出结果如：



原图



加权平均
木刻效果



算术平均

(4) 雕刻与浮雕

雕刻 (carve) 与浮雕 (relievo) 都是，利用每个像素与其左上方相邻像素的色差绝对值，来得到新图像的颜色值。为了防止图像太暗，给每个差值都加上灰度常数（如 128）；同时为了防止图像因差值过大而太刺眼，又要限制它们的最大差值（即最小值）（如 67）。可以修改这两个值来改变输出效果。

例如：

```

// 雕刻 (当前点 - 左上方点)
Color col, collt;
BYTE r, g, b;
pCarveImg = new Bitmap(w - 1, h - 1);
for (int i = 1; i < w; i++) {
    for (int j = 1; j < h; j++) {
        pImg->GetPixel(i, j, &col);
        pImg->GetPixel(i - 1, j - 1, &collt);
        r = BYTE(max(67, min(255, abs(col.GetR() - collt.GetR()) + 128)));
        g = BYTE(max(67, min(255, abs(col.GetG() - collt.GetG()) + 128)));
        b = BYTE(max(67, min(255, abs(col.GetB() - collt.GetB()) + 128)));
        col.SetValue(col.MakeARGB(255, r, g, b));
        pCarveImg->SetPixel(i - 1, j - 1, col);
    }
}
// 浮雕 (左上方点 - 当前点)
Color col, collt;
BYTE r, g, b;

```

```
pRelievoImg = new Bitmap(w - 1, h - 1);
for (int i = 1; i < w; i++) {
    for (int j = 1; j < h; j++) {
        pImg->GetPixel(i, j, &col);
        pImg->GetPixel(i - 1, j - 1, &collt);
        r = BYTE(max(67, min(255, abs(collt.GetR() - col.GetR() + 128))));
        g = BYTE(max(67, min(255, abs(collt.GetG() - col.GetG() + 128))));
        b = BYTE(max(67, min(255, abs(collt.GetB() - col.GetB() + 128))));
        col.SetValue(col.MakeARGB(255, r, g, b));
        pRelievoImg->SetPixel(i - 1, j - 1, col);
    }
}
```

输出结果如：



原图



雕刻



浮雕

雕刻与浮雕效果（最小值 67，差值常量 128）



雕刻



浮雕

雕刻与浮雕效果（最小值 50，差值常量 150）



8. 图元文件

从一开始 GDI 就支持(图)元文件(metafile)，早期(1985 年)的版本为 WMF(Windows MetaFile 视窗元文件)，主要针对 Win16 (Win3.x)，后来(1990 年)也支持 Win32 (Win95/98/Me)。以后(1993 年)随 Windows NT 推出了改进的元文件版本——EMF (Enhanced Windows MetaFile 增强型视窗元文件)，只支持 Win32 (Win95/98/Me/NT/2000/XP)。现在(2001 年)又随 GDI+推出了加强型 EMF——EMF+，可以同时支持 GDI 和 GDI+。

元文件所支持的 GDI 类型

元文件类型	Win16 GDI	Win32 GDI	Win32/64 GDI+
WMF	√	√	×
EMF	×	√	×
EMF+	×	√	√

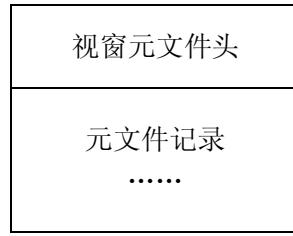
虽然在 GDI+中，将图元文件所对应的类 Metafile 作为 Image 的派生类，但这只是为了图元文件可以同时处理图形和图像。其实图元文件中所包含的就是一系列绘图(包括绘制图像)指令及参数，属于矢量图形文件。它所占空间小、可以任意缩放(不会产生马赛克效应)，但是绘制图形需要一定的时间。

1) 元文件格式

下面分别介绍 WMF、EMF 和 EMF+的文件格式。

(1) WMF 文件格式

- 元文件结构：



- 元文件头:

```
typedef struct tagMETAHEADER {
    WORD mtType; // 元文件类型: 内存=0、磁盘文件=1
    WORD mtHeaderSize; // 文件头大小的字数
    WORD mtVersion; // 系统的版本号: 支持 DIB=0x0300、不支持 DIB=0x0100
    DWORD mtSize; // 文件大小的字数
    WORD mtNoObjects; // 同时存在于元文件内的最大对象数
    DWORD mtMaxRecord; // 元文件中最大记录的字大小
    WORD mtNoParameters; // 保留字段
} METAHEADER;
```

- 元文件记录:

```
typedef struct tagMETARECORD
{
    DWORD      rdSize; // 记录大小的字数
    WORD       rdFunction; // 文件号 (记录类型 META_XXX)
    WORD       rdParm[1]; // 函数参数数组, 逆序排列
} METARECORD;
```

记录类型 (67 种)

记录	值	记录	值
META_SETBKCOLOR	0x0201	META_POLYGON	0x0324
META_SETBKMODE	0x0102	META_POLYLINE	0x0325
META_SETMAPMODE	0x0103	META_ESCAPE	0x0626
META_SETROP2	0x0104	META_RESTOREDC	0x0127
META_SETRELABS	0x0105	META_FILLREGION	0x0228
META_SETPOLYFILLMODE	0x0106	META_FRAMEREGION	0x0429
META_SETSTRETCHBLTMODE	0x0107	META_INVERTREGION	0x012A
META_SETTEXTCHAREXTRA	0x0108	META_PAINTREGION	0x012B
META_SETTEXTCOLOR	0x0209	META_SELECTCLIPREGION	0x012C
META_SETTEXTJUSTIFICATION	0x020A	META_SELECTOBJECT	0x012D
META_SETWINDOWORG	0x020B	META_SETTEXTALIGN	0x012E
META_SETWINDOWEXT	0x020C	META_CHORD	0x0830
META_SETVIEWPORTORG	0x020D	META_SETMAPPERFLAGS	0x0231
META_SETVIEWPORTEXT	0x020E	META_EXTTEXTOUT	0x0a32
META_OFFSETWINDOWORG	0x020F	META_SETDIBTODEV	0x0d33
META_SCALEWINDOWEXT	0x0410	META_SELECTPALETTE	0x0234

META_OFFSETVIEWPORTORG	0x0211	META_REALIZEPALETTE	0x0035
META_SCALEVIEWPORTEXT	0x0412	META_ANIMATEPALETTE	0x0436
META_LINETO	0x0213	META_SETPALENTRIES	0x0037
META_MOVETO	0x0214	META_POLYPOLYGON	0x0538
META_EXCLUDECLIPRECT	0x0415	META_RESIZEPALETTE	0x0139
META_INTERSECTCLIPRECT	0x0416	META_DIBBITBLT	0x0940
META_ARC	0x0817	META_DIBSTRETCHBLT	0x0b41
META_ELLIPSE	0x0418	META_DIBCREATEPATTERNBRUSH	0x0142
META_FLOODFILL	0x0419	META_STRETCHDIB	0x0f43
META_PIE	0x081A	META_EXTFLOODFILL	0x0548
META_RECTANGLE	0x041B	META_SETAYOUT	0x0149
META_ROUNDRECT	0x061C	META_DELETEOBJECT	0x01f0
META_PATBLT	0x061D	META_CREATEPALETTE	0x00f7
META_SAVEDC	0x001E	META_CREATEPATTERNBRUSH	0x01F9
META_SETPIXEL	0x041F	META_CREATEPENINDIRECT	0x02FA
META_OFFSETCLIPRGN	0x0220	META_CREATEFONTINDIRECT	0x02FB
META_TEXTOUT	0x0521	META_CREATEBRUSHINDIRECT	0x02FC
META_BITBLT	0x0922	META_CREATREGION	0x06FF
META_STRETCHBLT	0x0B23		

其中的 META_SETAYOUT 只适用于 Win98/Win2000 以上

(2) EMF 文件格式

- 元文件结构:



- 元文件头:

```

typedef struct tagENHMETAHEADER {
    DWORD iType; // 记录类型, 必须 = EMR_HEADER (= 1)
    DWORD nSize; // 结构的字节大小, 可能 > sizeof(ENHMETAHEADER)
    RECTL rclBounds; // 边界矩形 (设备单位, 含右边和底边)
    RECTL rclFrame; // 边界矩形 (0.01 毫米单位 HIMETRIC, 含右边和底边)
    DWORD dSignature; // 签名, 必须 = ENHMETA_SIGNATURE (= 0x464D4520)
}
  
```

```

DWORD nVersion; // 元文件版本, 当前 = 0x10000
DWORD nBytes; // 元文件的字节大小
DWORD nRecords; // 元文件中的记录数
WORD nHandles; // 元文件句柄表中的句柄数 (第 0 个句柄被保留)
WORD sReserved; // 保留, 必须 = 0
DWORD nDescription; // 描述字符串中的字符数, 无描述串时必须设为 0
DWORD offDescription; // 描述串相对于文件头开始处的偏移量, 无描述串时必须设为 0
DWORD nPalEntries; // 元文件内调色板中的表项数
SIZEL szlDevice; // 以像素为单位的参考设备分辨率
SIZEL szlMillimeters; // 以毫米为单位的参考设备分辨率
#if (WINVER >= 0x0400) // Win95/WinNT4.0 以上
    DWORD cbPixelFormat; // 像素格式, 无像素格式设置时 = 0,
        // 开始设置为 DC 时 = sizeOf(PIXELFORMATDESCRIPTOR)、
        // 有多个单像素格式设置时 = 指向最后一个像素格式头的指针
    DWORD offPixelFormat; // 像素格式的偏移量
    DWORD bOpenGL; // 元文件中是否包含 OpenGL 记录
#endif /* WINVER >= 0x0400 */
#if (WINVER >= 0x0500) // Win98/Win2000 以上
    SIZEL szlMicrometers; // 以微米表示的参考设备大小
#endif /* WINVER >= 0x0500 */
} ENHMETAHEADER;

```

其中：

```

typedef struct _RECTL
{
    LONG    left;
    LONG    top;
    LONG    right;
    LONG    bottom;
} RECTL;

typedef SIZE SIZEL;
typedef struct tagSIZE
{
    LONG    cx;
    LONG    cy;
} SIZE;

```

- 元文件记录：

```

typedef struct tagENHMETARECORD {
    DWORD iType; // 记录类型
    DWORD nSize; // 记录的字节大小
    DWORD dParm[1]; // 传递给 GDI 函数的参数数组
} ENHMETARECORD;

```

记录类型（121 种，字母序）

记录	值	记录	值
EMR_ABORTPATH	68	EMR_POLYLINE	4
EMR_ANGLEARC	41	EMR_POLYLINE16	87
EMR_ARC	45	EMR_POLYLINETO	6
EMR_ARCTO	55	EMR_POLYLINETO16	89
EMR_BEGINPATH	59	EMR_POLYPOLYGON	8
EMR_BITBLT	76	EMR_POLYPOLYGON16	91
EMR_CHORD	46	EMR_POLYPOLYLINE	7
EMR_CLOSEFIGURE	61	EMR_POLYPOLYLINE16	90
EMR_CREATEBRUSHINDIRECT	39	EMR_POLYTEXTOUTA	96
EMR_CREATEDIBPATTERNBRUSHPT	94	EMR_POLYTEXTOUTW	97
EMR_CREATEMONOBRUSH	93	EMR_REALIZEPALETTE	52
EMR_CREATEPALETTE	49	EMR_RECTANGLE	43
EMR_CREATEPEN	38	EMR_RESIZEPALETTE	51
EMR_DELETEOBJECT	40	EMR_RESTOREDC	34
EMR_ELLIPSE	42	EMR_ROUNDRECT	44
EMR_ENDPATH	60	EMR_SAVEDC	33
EMR_EOF	14	EMR_SCALEVIEWPORTEXTEX	31
EMR_EXCLUDECLIPRECT	29	EMR_SCALEWINDOWEXTEX	32
EMR_EXTCREATEFONTINDIRECTW	82	EMR_SELECTCLIPPATH	67
EMR_EXTCREATEPEN	95	EMR_SELECTOBJECT	37
EMR_EXTFLOODFILL	53	EMR_SELECTPALETTE	48
EMR_EXTSELECTCLIPRGN	75	EMR_SETARCDIRECTION	57
EMR_EXTTXTOUTA	83	EMR_SETBKCOLOR	25
EMR_EXTTXTOUTW	84	EMR_SETBKMODE	18
EMR_FILLPATH	62	EMR_SETBRUSHORGEX	13
EMR_FILLRGN	71	EMR_SETCOLORADJUSTMENT	23
EMR_FLATTENPATH	65	EMR_SETDIBITSTODEVICE	80
EMR_FRAMERGN	72	EMR_SETMAPMODE	17
EMR_GDICOMMENT	70	EMR_SETMAPPERFLAGS	16
EMR_HEADER	1	EMR_SETMETARGN	28
EMR_INTERSECTCLIPRECT	30	EMR_SETMITERLIMIT	58
EMR_INVERTRGN	73	EMR_SETPALETTEENTRIES	50
EMR_LINETO	54	EMR_SETPIXELV	15
EMR_MASKBLT	78	EMR_SETPOLYFILLMODE	19
EMR MODIFYWORLDTRANSFORM	36	EMR_SETROP2	20
EMR_MOVETOEX	27	EMR_SETSTRETCHBLTMODE	21
EMR_OFFSETCLIPRGN	26	EMR_SETTEXTALIGN	22
EMR_PAINTRGN	74	EMR_SETTEXTCOLOR	24
EMR_PIE	47	EMR_SETVIEWPORTEXTEX	11
EMR_PLGBLT	79	EMR_SETVIEWPORTORGEX	12
EMR_POLYBEZIER	2	EMR_SETWINDOWEXTEX	9

EMR_POLYBEZIER16	85	EMR_SETWINDOWORGEX	10
EMR_POLYBEZIERTO	5	EMR_SETWORLDTRANSFORM	35
EMR_POLYBEZIERTO16	88	EMR_STRETCHBLT	77
EMR_POLYDRAW	56	EMR_STRETCHDIBITS	81
EMR_POLYDRAW16	92	EMR_STROKEANDFILLPATH	63
EMR_POLYGON	3	EMR_STROKEPATH	64
EMR_POLYGON16	86	EMR_WIDENPATH	66

记录类型 (121 种, 数值序)

记录	值	记录	值
EMR_HEADER	1	EMR_FILLPATH	62
EMR_POLYBEZIER	2	EMR_STROKEANDFILLPATH	63
EMR_POLYGON	3	EMR_STROKEPATH	64
EMR_POLYLINE	4	EMR_FLATTENPATH	65
EMR_POLYBEZIERTO	5	EMR_WIDENPATH	66
EMR_POLYLINETO	6	EMR_SELECTCLIPPATH	67
EMR_POLYPOLYLINE	7	EMR_ABORTPATH	68
EMR_POLYPOLYGON	8		
EMR_SETWINDOWTEXTEX	9	EMR_GDICOMMENT	70
EMR_SETWINDOWORGEX	10	EMR_FILLRGN	71
EMR_SETVIEWPORTEXTEX	11	EMR_FRAMERGN	72
EMR_SETVIEWPORTORGEX	12	EMR_INVERTRGN	73
EMR_SETBRUSHORGEX	13	EMR_PAINTRGN	74
EMR_EOF	14	EMR_EXTSELECTCLIPRGN	75
EMR_SETPIXELV	15	EMR_BITBLT	76
EMR_SETMAPPERFLAGS	16	EMR_STRETCHBLT	77
EMR_SETMAPMODE	17	EMR_MASKBLT	78
EMR_SETBKMODE	18	EMR_PLGBLT	79
EMR_SETPOLYFILLMODE	19	EMR_SETDIBITSTODEVICE	80
EMR_SETROP2	20	EMR_STRETCHDIBITS	81
EMR_SETSTRETCHBLTMODE	21	EMR_EXTCREATEFONTINDIRECTW	82
EMR_SETTEXTALIGN	22	EMR_EXTEXTOUTA	83
EMR_SETCOLORADJUSTMENT	23	EMR_EXTEXTOUTW	84
EMR_SETTEXTCOLOR	24	EMR_POLYBEZIER16	85
EMR_SETBKCOLOR	25	EMR_POLYGON16	86
EMR_OFFSETCLIPRGN	26	EMR_POLYLINE16	87
EMR_MOVETOEX	27	EMR_POLYBEZIERTO16	88
EMR_SETMETARGN	28	EMR_POLYLINETO16	89
EMR_EXCLUDECLIPRECT	29	EMR_POLYPOLYLINE16	90
EMR_INTERSECTCLIPRECT	30	EMR_POLYPOLYGON16	91
EMR_SCALEVIEWPORTEXTEX	31	EMR_POLYDRAW16	92
EMR_SCALEWINDOWTEXTEX	32	EMR_CREATEMONOBRUSH	93
EMR_SAVEDC	33	EMR_CREATEDIBPATTERNBRUSHPT	94

EMR_RESTOREDC	34	EMR_EXTCREATEPEN	95
EMR_SETWORLDTRANSFORM	35	EMR_POLYTEXTOUTA	96
EMR MODIFYWORLDTRANSFORM	36	EMR_POLYTEXTOUTW	97
EMR_SELECTOBJECT	37	EMR_SETICMMODE	98
EMR_CREATEPEN	38	EMR_CREATECOLORSPACE	99
EMR_CREATEBRUSHINDIRECT	39	EMR_SETCOLORSPACE	100
EMR_DELETEOBJECT	40	EMR_DELETECOLORSPACE	101
EMR_ANGLEARC	41	EMR_GLSRECORD	102
EMR_ELLIPSE	42	EMR_GLSBOUNDEDRECORD	103
EMR_RECTANGLE	43	EMR_PIXELFORMAT	104
EMR_ROUNDRECT	44	EMR_RESERVED_105	105
EMR_ARC	45	EMR_RESERVED_106	106
EMR_CHORD	46	EMR_RESERVED_107	107
EMR_PIE	47	EMR_RESERVED_108	108
EMR_SELECTPALETTE	48	EMR_RESERVED_109	109
EMR_CREATEPALETTE	49	EMR_RESERVED_110	110
EMR_SETPALETTEENTRIES	50	EMR_COLORCORRECTPALETTE	111
EMR_RESIZEPALETTE	51	EMR_SETICMPFILEA	112
EMR_REALIZEPALETTE	52	EMR_SETICMPFILEW	113
EMR_EXTFLOODFILL	53	EMR_ALPHABLEND	114
EMR_LINETO	54	EMR_SETAYOUT	115
EMR_ARCTO	55	EMR_TRANSPARENTBLT	116
EMR_POLYDRAW	56	EMR_RESERVED_117	117
EMR_SETARCDIRECTION	57	EMR_GRADIENTFILL	118
EMR_SETMITERLIMIT	58	EMR_RESERVED_119	119
EMR_BEGINPATH	59	EMR_RESERVED_120	120
EMR_ENDPATH	60	EMR_COLORMATCHTOTARGETW	121
EMR_CLOSEFIGURE	61	EMR_CREATECOLORSPACEW	122

其中，69 空缺、98~104 需 Win95/WinNT4.0 以上系统、105~122 需 Win98/Win2000 以上系统

(3) EMF+文件格式

- 元文件结构和元文件记录同 EMF 的
- 元文件头：（去掉了 EMF 文件头的最后四个字段）

```

typedef struct {
    DWORD iType; // 记录类型，必须 =EMR_HEADER (= 1)
    DWORD nSize; // 结构的字节大小，可能 > sizeof(ENHMETAHEADER)
    RECTL rclBounds; // 边界矩形 (设备单位，含右边和底边)
    RECTL rclFrame; // 边界矩形 (0.01 毫米单位，含右边和底边)
    DWORD dSignature; // 签名，必须 =ENHMETA_SIGNATURE (= 0x464D4520)
    DWORD nVersion; // 元文件版本，当前 =0x10000
    DWORD nBytes; // 元文件的字节大小
    DWORD nRecords; // 元文件中的记录数
}

```

```

WORD nHandles; // 元文件句柄表中的句柄数（第 0 个句柄被保留）
WORD sReserved; // 保留，必须 = 0
DWORD nDescription; // 描述字符数组中的字符数，无描述串时必须设为 0
DWORD offDescription; // 描述串相对于文件头开始处的偏移量，无描述串时必须设为 0
DWORD nPalEntries; // 元文件内调色板中的表项数
SIZEL szlDevice; // 以像素为单位的参考设备分辨率
SIZEL szlMillimeters; // 以毫米为单位的参考设备分辨率
} ENHMETAHEADER3;

```

```

typedef enum { // 共 266 种
    // WMF 记录类型 (79 种)
    WmfRecordTypeSetBkColor = GDIP_WMF_RECORD_TO_EMFPLUS(META_SETBKCOLOR),
    WmfRecordTypeSetBkMode = GDIP_WMF_RECORD_TO_EMFPLUS(META_SETBKMODE),
    WmfRecordTypeSetMapMode = GDIP_WMF_RECORD_TO_EMFPLUS(META_SETMAPMODE),
    WmfRecordTypeSetROP2 = GDIP_WMF_RECORD_TO_EMFPLUS(META_SETROP2),
    WmfRecordTypeSetRelAbs = GDIP_WMF_RECORD_TO_EMFPLUS(META_SETRELABS),
    WmfRecordTypeSetPolyFillMode = GDIP_WMF_RECORD_TO_EMFPLUS(META_SETPOLYFILLMODE),
    WmfRecordTypeSetStretchBltMode = GDIP_WMF_RECORD_TO_EMFPLUS(META_SETSTRETCHBLTMODE),
    WmfRecordTypeSetTextCharExtra = GDIP_WMF_RECORD_TO_EMFPLUS(META_SETTEXTCHAREXTRA),
    WmfRecordTypeSetTextColor = GDIP_WMF_RECORD_TO_EMFPLUS(META_SETTEXTCOLOR),
    WmfRecordTypeSetTextJustification = GDIP_WMF_RECORD_TO_EMFPLUS(META_SETTEXTJUSTIFICATION),
    WmfRecordTypeSetWindowOrg = GDIP_WMF_RECORD_TO_EMFPLUS(META_SETWINDOWORG),
    WmfRecordTypeSetWindowExt = GDIP_WMF_RECORD_TO_EMFPLUS(META_SETWINDOWEXT),
    WmfRecordTypeSetViewportOrg = GDIP_WMF_RECORD_TO_EMFPLUS(META_SETVIEWPORTORG),
    WmfRecordTypeSetViewportExt = GDIP_WMF_RECORD_TO_EMFPLUS(META_SETVIEWPORTEXT),
    WmfRecordTypeOffsetWindowOrg = GDIP_WMF_RECORD_TO_EMFPLUS(META_OFFSETWINDOWORG),
    WmfRecordTypeScaleWindowExt = GDIP_WMF_RECORD_TO_EMFPLUS(META_SCALEWINDOWEXT),
    WmfRecordTypeOffsetViewportOrg = GDIP_WMF_RECORD_TO_EMFPLUS(META_OFFSETVIEWPORTORG),
    WmfRecordTypeScaleViewportExt = GDIP_WMF_RECORD_TO_EMFPLUS(META_SCALEVIEWPORTEXT),
    WmfRecordTypeLineTo = GDIP_WMF_RECORD_TO_EMFPLUS(META_LINETO),
    WmfRecordTypeMoveTo = GDIP_WMF_RECORD_TO_EMFPLUS(META_MOVETO),
    WmfRecordTypeExcludeClipRect = GDIP_WMF_RECORD_TO_EMFPLUS(META_EXCLUDECLIPRECT),
    WmfRecordTypeIntersectClipRect = GDIP_WMF_RECORD_TO_EMFPLUS(META_INTERSECTCLIPRECT),
    WmfRecordTypeArc = GDIP_WMF_RECORD_TO_EMFPLUS(META_ARC),
    WmfRecordTypeEllipse = GDIP_WMF_RECORD_TO_EMFPLUS(META_ELLIPSE),
    WmfRecordTypeFloodFill = GDIP_WMF_RECORD_TO_EMFPLUS(META_FLOODFILL),
    WmfRecordTypePie = GDIP_WMF_RECORD_TO_EMFPLUS(META_PIE),
    WmfRecordTypeRectangle = GDIP_WMF_RECORD_TO_EMFPLUS(META_RECTANGLE),
    WmfRecordTypeRoundRect = GDIP_WMF_RECORD_TO_EMFPLUS(META_ROUNDRECT),
    WmfRecordTypePatBlt = GDIP_WMF_RECORD_TO_EMFPLUS(META_PATBLT),
    WmfRecordTypeSaveDC = GDIP_WMF_RECORD_TO_EMFPLUS(META_SAVEDC),
    WmfRecordTypeSetPixel = GDIP_WMF_RECORD_TO_EMFPLUS(META_SETPIXEL),
    WmfRecordTypeOffsetClipRgn = GDIP_WMF_RECORD_TO_EMFPLUS(META_OFFSETCLIPRGN),
    WmfRecordTypeTextOut = GDIP_WMF_RECORD_TO_EMFPLUS(META_TEXTOUT),
}

```

WmfRecordTypeBitBlt = GDIP_WMF_RECORD_TO_EMFPLUS(META_BITBLT),
WmfRecordTypeStretchBlt = GDIP_WMF_RECORD_TO_EMFPLUS(META_STRETCHBLT),
WmfRecordTypePolygon = GDIP_WMF_RECORD_TO_EMFPLUS(META_POLYGON),
WmfRecordTypePolyline = GDIP_WMF_RECORD_TO_EMFPLUS(META_POLYLINE),
WmfRecordTypeEscape = GDIP_WMF_RECORD_TO_EMFPLUS(META_ESCAPE),
WmfRecordTypeRestoreDC = GDIP_WMF_RECORD_TO_EMFPLUS(META_RESTOREDC),
WmfRecordTypeFillRegion = GDIP_WMF_RECORD_TO_EMFPLUS(META_FILLREGION),
WmfRecordTypeFrameRegion = GDIP_WMF_RECORD_TO_EMFPLUS(META_FRAMEREGION),
WmfRecordTypeInvertRegion = GDIP_WMF_RECORD_TO_EMFPLUS(META_INVERTREGION),
WmfRecordTypePaintRegion = GDIP_WMF_RECORD_TO_EMFPLUS(META_PAINTREGION),
WmfRecordTypeSelectClipRegion = GDIP_WMF_RECORD_TO_EMFPLUS(META_SELECTCLIPREGION),
WmfRecordTypeSelectObject = GDIP_WMF_RECORD_TO_EMFPLUS(META_SELECTOBJECT),
WmfRecordTypeSetTextAlign = GDIP_WMF_RECORD_TO_EMFPLUS(META_SETTEXTALIGN),
WmfRecordTypeDrawText = GDIP_WMF_RECORD_TO_EMFPLUS(0x062F),
WmfRecordTypeChord = GDIP_WMF_RECORD_TO_EMFPLUS(META_CHORD),
WmfRecordTypeSetMapperFlags = GDIP_WMF_RECORD_TO_EMFPLUS(META_SETMAPPERFLAGS),
WmfRecordTypeExtTextOut = GDIP_WMF_RECORD_TO_EMFPLUS(META_EXTTEXTOUT),
WmfRecordTypeSetDIBToDev = GDIP_WMF_RECORD_TO_EMFPLUS(META_SETDIBTODEV),
WmfRecordTypeSelectPalette = GDIP_WMF_RECORD_TO_EMFPLUS(META_SELECTPALETTE),
WmfRecordTypeRealizePalette = GDIP_WMF_RECORD_TO_EMFPLUS(META_REALIZEPALETTE),
WmfRecordTypeAnimatePalette = GDIP_WMF_RECORD_TO_EMFPLUS(META_ANIMATEPALETTE),
WmfRecordTypeSetPalEntries = GDIP_WMF_RECORD_TO_EMFPLUS(META_SETPALENTRIES),
WmfRecordTypePolyPolygon = GDIP_WMF_RECORD_TO_EMFPLUS(META_POLYPOLYGON),
WmfRecordTypeResizePalette = GDIP_WMF_RECORD_TO_EMFPLUS(META_RESIZEPALETTE),
WmfRecordTypeDIBBitBlt = GDIP_WMF_RECORD_TO_EMFPLUS(META_DIBBITBLT),
WmfRecordTypeDIBStretchBlt = GDIP_WMF_RECORD_TO_EMFPLUS(META_DIBSTRETCHBLT),
WmfRecordTypeDIBCreatePatternBrush = GDIP_WMF_RECORD_TO_EMFPLUS(META_DIBCREATEPATTERNBRUSH),
WmfRecordTypeStretchDIB = GDIP_WMF_RECORD_TO_EMFPLUS(META_STRETCHDIB),
WmfRecordTypeExtFloodFill = GDIP_WMF_RECORD_TO_EMFPLUS(META_EXTFLOODFILL),
WmfRecordTypeSetLayout = GDIP_WMF_RECORD_TO_EMFPLUS(0x0149),
WmfRecordTypeResetDC = GDIP_WMF_RECORD_TO_EMFPLUS(0x014C),
WmfRecordTypeStartDoc = GDIP_WMF_RECORD_TO_EMFPLUS(0x014D),
WmfRecordTypeStartPage = GDIP_WMF_RECORD_TO_EMFPLUS(0x004F),
WmfRecordTypeEndPage = GDIP_WMF_RECORD_TO_EMFPLUS(0x0050),
WmfRecordTypeAbortDoc = GDIP_WMF_RECORD_TO_EMFPLUS(0x0052),
WmfRecordTypeEndDoc = GDIP_WMF_RECORD_TO_EMFPLUS(0x005E),
WmfRecordTypeDeleteObject = GDIP_WMF_RECORD_TO_EMFPLUS(META_DELETEOBJECT),
WmfRecordTypeCreatePalette = GDIP_WMF_RECORD_TO_EMFPLUS(META_CREATEPALETTE),
WmfRecordTypeCreateBrush = GDIP_WMF_RECORD_TO_EMFPLUS(0x00F8),
WmfRecordTypeCreatePatternBrush = GDIP_WMF_RECORD_TO_EMFPLUS(META_CREATEPATTERNBRUSH),
WmfRecordTypeCreatePenIndirect = GDIP_WMF_RECORD_TO_EMFPLUS(META_CREATEPENINDIRECT),
WmfRecordTypeCreateFontIndirect = GDIP_WMF_RECORD_TO_EMFPLUS(META_CREATEFONTINDIRECT),
WmfRecordTypeCreateBrushIndirect = GDIP_WMF_RECORD_TO_EMFPLUS(META_CREATEBRUSHINDIRECT),
WmfRecordTypeCreateBitmapIndirect = GDIP_WMF_RECORD_TO_EMFPLUS(0x02FD),

```
WmfRecordTypeCreateBitmap = GDIP_WMF_RECORD_TO_EMFPLUS(0x06FE),
WmfRecordTypeCreateRegion = GDIP_WMF_RECORD_TO_EMFPLUS(META_CREATEREGION),
// EMF 记录类型 (124 种)
EmfRecordTypeHeader = EMR_HEADER,
EmfRecordTypePolyBezier = EMR_POLYBEZIER,
EmfRecordTypePolygon = EMR_POLYGON,
EmfRecordTypePolyline = EMR_POLYLINE,
EmfRecordTypePolyBezierTo = EMR_POLYBEZIERTO,
EmfRecordTypePolyLineTo = EMR_POLYLINETO,
EmfRecordTypePolyPolyline = EMR_POLYPOLYLINE,
EmfRecordTypePolyPolygon = EMR_POLYPOLYGON,
EmfRecordTypeSetWindowExtEx = EMR_SETWINDOWEXTEX,
EmfRecordTypeSetWindowOrgEx = EMR_SETWINDOWORGEX,
EmfRecordTypeSetViewportExtEx = EMR_SETVIEWPORTEXTEX,
EmfRecordTypeSetViewportOrgEx = EMR_SETVIEWPORTORGEX,
EmfRecordTypeSetBrushOrgEx = EMR_SETBRUSHORGEX,
EmfRecordTypeEOF = EMR_EOF,
EmfRecordTypeSetPixelV = EMR_SETPIXELV,
EmfRecordTypeSetMapperFlags = EMR_SETMAPPERFLAGS,
EmfRecordTypeSetMapMode = EMR_SETMAPMODE,
EmfRecordTypeSetBkMode = EMR_SETBKMODE,
EmfRecordTypeSetPolyFillMode = EMR_SETPOLYFILLMODE,
EmfRecordTypeSetROP2 = EMR_SETROP2,
EmfRecordTypeSetStretchBltMode = EMR_SETSTRETCHBLTMODE,
EmfRecordTypeSetTextAlign = EMR_SETTEXTALIGN,
EmfRecordTypeSetColorAdjustment = EMR_SETCOLORADJUSTMENT,
EmfRecordTypeSetTextColor = EMR_SETTEXTCOLOR,
EmfRecordTypeSetBkColor = EMR_SETBKCOLOR,
EmfRecordTypeOffsetClipRgn = EMR_OFFSETCLIPRGN,
EmfRecordTypeMoveToEx = EMR_MOVETOEX,
EmfRecordTypeSetMetaRgn = EMR_SETMETARGN,
EmfRecordTypeExcludeClipRect = EMR_EXCLUDECLIPRECT,
EmfRecordTypeIntersectClipRect = EMR_INTERSECTCLIPRECT,
EmfRecordTypeScaleViewportExtEx = EMR_SCALEVIEWPORTEXTEX,
EmfRecordTypeScaleWindowExtEx = EMR_SCALEWINDOWEXTEX,
EmfRecordTypeSaveDC = EMR_SAVEDC,
EmfRecordTypeRestoreDC = EMR_RESTOREDC,
EmfRecordTypeSetWorldTransform = EMR_SETWORLDTRANSFORM,
EmfRecordTypeModifyWorldTransform = EMR_MODIFYWORLDTRANSFORM,
EmfRecordTypeSelectObject = EMR_SELECTOBJECT,
EmfRecordTypeCreatePen = EMR_CREATEPEN,
EmfRecordTypeCreateBrushIndirect = EMR_CREATEBRUSHINDIRECT,
EmfRecordTypeDeleteObject = EMR_DELETEOBJECT,
EmfRecordTypeAngleArc = EMR_ANGLEARC,
```

EmfRecordTypeEllipse = EMR_ELLIPSE,
EmfRecordTypeRectangle = EMR_RECTANGLE,
EmfRecordTypeRoundRect = EMR_ROUNDRECT,
EmfRecordTypeArc = EMR_ARC,
EmfRecordTypeChord = EMR_CHORD,
EmfRecordTypePie = EMR_PIE,
EmfRecordTypeSelectPalette = EMR_SELECTPALETTE,
EmfRecordTypeCreatePalette = EMR_CREATEPALETTE,
EmfRecordTypeSetPaletteEntries = EMR_SETPALETTEENTRIES,
EmfRecordTypeResizePalette = EMR_RESIZEPALETTE,
EmfRecordTypeRealizePalette = EMR_REALIZEPALETTE,
EmfRecordTypeExtFloodFill = EMR_EXTFLOODFILL,
EmfRecordTypeLineTo = EMR_LINETO,
EmfRecordTypeArcTo = EMR_ARCTO,
EmfRecordTypePolyDraw = EMR_POLYDRAW,
EmfRecordTypeSetArcDirection = EMR_SETARCDIRECTION,
EmfRecordTypeSetMiterLimit = EMR_SETMITERLIMIT,
EmfRecordTypeBeginPath = EMR_BEGINPATH,
EmfRecordTypeEndPath = EMR_ENDPATH,
EmfRecordTypeCloseFigure = EMR_CLOSEFIGURE,
EmfRecordTypeFillPath = EMR_FILLPATH,
EmfRecordTypeStrokeAndFillPath = EMR_STROKEANDFILLPATH,
EmfRecordTypeStrokePath = EMR_STROKEPATH,
EmfRecordTypeFlattenPath = EMR_FLATTENPATH,
EmfRecordTypeWidenPath = EMR_WIDENPATH,
EmfRecordTypeSelectClipPath = EMR_SELECTCLIPPATH,
EmfRecordTypeAbortPath = EMR_ABORTPATH,
EmfRecordTypeReserved_069 = 69,
EmfRecordTypeGdiComment = EMR_GDICOMMENT,
EmfRecordTypeFillRgn = EMR_FILLRGN,
EmfRecordTypeFrameRgn = EMR_FRAME RGN,
EmfRecordTypeInvertRgn = EMR_INVERTRGN,
EmfRecordTypePaintRgn = EMR_PAINTRGN,
EmfRecordTypeExtSelectClipRgn = EMR_EXTSELECTCLIPRGN,
EmfRecordTypeBitBlt = EMR_BITBLT,
EmfRecordTypeStretchBlt = EMR_STRETCHBLT,
EmfRecordTypeMaskBlt = EMR_MASKBLT,
EmfRecordTypePlgBlt = EMR_PLGBLT,
EmfRecordTypeSetDIBitsToDevice = EMR_SETDIBITSTODEVIC,
EmfRecordTypeStretchDIBits = EMR_STRETCHDIBITS,
EmfRecordTypeExtCreateFontIndirect = EMR_EXTCREATEFONTINDIRECTW,
EmfRecordTypeExtTextOutA = EMR_EXTTEXTOUTA,
EmfRecordTypeExtTextOutW = EMR_EXTTEXTOUTW,
EmfRecordTypePolyBezier16 = EMR_POLYBEZIER16,

```
EmfRecordTypePolygon16 = EMR_POLYGON16,
EmfRecordTypePolyline16 = EMR_POLYLINE16,
EmfRecordTypePolyBezierTo16 = EMR_POLYBEZIERTO16,
EmfRecordTypePolylineTo16 = EMR_POLYLINETO16,
EmfRecordTypePolyPolyline16 = EMR_POLYPOLYLINE16,
EmfRecordTypePolyPolygon16 = EMR_POLYPOLYGON16,
EmfRecordTypePolyDraw16 = EMR_POLYDRAW16,
EmfRecordTypeCreateMonoBrush = EMR_CREATEMONOBRUSH,
EmfRecordTypeCreateDIBPatternBrushPt = EMR_CREATEDIBPATTERNBRUSHPT,
EmfRecordTypeExtCreatePen = EMR_EXTCREATEPEN,
EmfRecordTypePolyTextOutA = EMR_POLYTEXTOUTA,
EmfRecordTypePolyTextOutW = EMR_POLYTEXTOUTW,
EmfRecordTypeSetICMMode = 98,
EmfRecordTypeCreateColorSpace = 99,
EmfRecordTypeSetColorSpace = 100,
EmfRecordTypeDeleteColorSpace = 101,
EmfRecordTypeGLSRecord = 102,
EmfRecordTypeGLSBoundedRecord = 103,
EmfRecordTypePixelFormat = 104,
EmfRecordTypeDrawEscape = 105,
EmfRecordTypeExtEscape = 106,
EmfRecordTypeStartDoc = 107,
EmfRecordTypeSmallTextOut = 108,
EmfRecordTypeForceUFIMapping = 109,
EmfRecordTypeNamedEscape = 110,
EmfRecordTypeColorCorrectPalette = 111,
EmfRecordTypeSetICMPProfileA = 112,
EmfRecordTypeSetICMPProfileW = 113,
EmfRecordTypeAlphaBlend = 114,
EmfRecordTypeSetLayout = 115,
EmfRecordTypeTransparentBlt = 116,
EmfRecordTypeReserved_117 = 117,
EmfRecordTypeGradientFill = 118,
EmfRecordTypeSetLinkedUFI = 119,
EmfRecordTypeSetTextJustification = 120,
EmfRecordTypeColorMatchToTargetW = 121,
EmfRecordTypeCreateColorSpaceW = 122,
EmfRecordTypeMax = 122,
EmfRecordTypeMin = 1,
// EMF+记录类型（63 种）
EmfPlusRecordTypeInvalid = GDIP_EMFPLUS_RECORD_BASE,
EmfPlusRecordTypeHeader,
EmfPlusRecordTypeEndOfFile,
EmfPlusRecordTypeComment,
```

EmfPlusRecordTypeGetDC,
EmfPlusRecordTypeMultiFormatStart,
EmfPlusRecordTypeMultiFormatSection,
EmfPlusRecordTypeMultiFormatEnd,
EmfPlusRecordTypeObject,
EmfPlusRecordTypeClear,
EmfPlusRecordTypeFillRects,
EmfPlusRecordTypeDrawRects,
EmfPlusRecordTypeFillPolygon,
EmfPlusRecordTypeDrawLines,
EmfPlusRecordTypeFillEllipse,
EmfPlusRecordTypeDrawEllipse,
EmfPlusRecordTypeFillPie,
EmfPlusRecordTypeDrawPie,
EmfPlusRecordTypeDrawArc,
EmfPlusRecordTypeFillRegion,
EmfPlusRecordTypeFillPath,
EmfPlusRecordTypeDrawPath,
EmfPlusRecordTypeFillClosedCurve,
EmfPlusRecordTypeDrawClosedCurve,
EmfPlusRecordTypeDrawCurve,
EmfPlusRecordTypeDrawBeziers,
EmfPlusRecordTypeDrawImage,
EmfPlusRecordTypeDrawImagePoints,
EmfPlusRecordTypeDrawString,
EmfPlusRecordTypeSetRenderingOrigin,
EmfPlusRecordTypeSetAntiAliasMode,
EmfPlusRecordTypeSetTextRenderingHint,
EmfPlusRecordTypeSetTextContrast,
EmfPlusRecordTypeSetGammaValue,
EmfPlusRecordTypeSetInterpolationMode,
EmfPlusRecordTypeSetPixelOffsetMode,
EmfPlusRecordTypeSetCompositingMode,
EmfPlusRecordTypeSetCompositingQuality,
EmfPlusRecordTypeSave,
EmfPlusRecordTypeRestore,
EmfPlusRecordTypeBeginContainer,
EmfPlusRecordTypeBeginContainerNoParams,
EmfPlusRecordTypeEndContainer,
EmfPlusRecordTypeSetWorldTransform,
EmfPlusRecordTypeResetWorldTransform,
EmfPlusRecordTypeMultiplyWorldTransform,
EmfPlusRecordTypeTranslateWorldTransform,
EmfPlusRecordTypeScaleWorldTransform,

```

EmfPlusRecordTypeRotateWorldTransform,
EmfPlusRecordTypeSetPageTransform,
EmfPlusRecordTypeResetClip,
EmfPlusRecordTypeSetClipRect,
EmfPlusRecordTypeSetClipPath,
EmfPlusRecordTypeSetClipRegion,
EmfPlusRecordTypeOffsetClip,
EmfPlusRecordTypeDrawDriverString,
EmfPlusRecordTypeStrokeFillPath,
EmfPlusRecordTypeSerializableObject,
EmfPlusRecordTypeSetTSGraphics,
EmfPlusRecordTypeSetTSClip,
EmfPlusRecordTotal,
EmfPlusRecordTypeMax = EmfPlusRecordTotal-1,
EmfPlusRecordTypeMin = EmfPlusRecordTypeHeader
} EmfPlusRecordType;

```

其中：

```

#define GDIP_EMFPLUS_RECORD_BASE      0x00004000
#define GDIP_WMF_RECORD_BASE         0x00010000
#define GDIP_WMF_RECORD_TO_EMFPLUS(n) ((EmfPlusRecordType)((n) | GDIP_WMF_RECORD_BASE))
#define GDIP_EMFPLUS_RECORD_TO_WMF(n)  ((n) & (~GDIP_WMF_RECORD_BASE))
#define GDIP_IS_WMF_RECORDTYPE(n)     (((n) & GDIP_WMF_RECORD_BASE) != 0)

```

2) 在 GDI 中使用图元文件 (SDK)

在 GDI 中使用图元文件，就是先用 `Create*MetaFile` 函数创建图元文件，同时获取对应的元文件 DC 句柄；然后利用 SDK 的各种绘图（工具）函数（一般与 CDC 类的同功能函数同名，只是多一个 HDC 参数，作为第一个输入参数，这里使用的是元文件 DC）向元文件（DC）添加记录；最后利用 `Play*MetaFile` 函数显示元文件（重绘文件中的图形记录）。

下面分别介绍在 GDI 中使用 WMF 和 EMF 图元文件具体方法。下面讨论都是基于 SDK 的，所用到的各种函数也都是 SDK 中的全局函数。

(1) WMF

主要 SDK 函数有：

```

HDC CreateMetaFile(LPCTSTR lpszFile); // 创建图元文件
// lpszFile=NULL 时，创建内存图元文件
HMETAFILE CloseMetaFile(HDC hdc); // 关闭图元文件
HMETAFILE GetMetaFile(LPCWSTR lpName); // 装入图元文件
BOOL PlayMetaFile(HDC hdc, HMETAFILE hmf); // 显示图元文件
BOOL DeleteMetaFile(HMETAFILE hmf); // 删除内存图元文件或磁盘图文件的句柄

```

例如：

```

// 创建图元文件
HDC hdc = CreateMetaFile(L"test.wmf"); // 创建 WMF 文件
Ellipse(hdc, 10, 10, 150, 100); // 绘制图形
Rectangle(hdc, 160, 100, 360, 200);
HMETAFILE hwmf = CloseMetaFile(hdc); // 关闭图元文件

// 装入并显示图元文件
//HMETAFILE hwmf = GetMetaFile(L"test.wmf");
if (hwmf != NULL) {
    PlayMetaFile(GetDC()->m_hDC, hwmf);
    DeleteMetaFile(hwmf);
}

```

其他常用函数有：

```

BOOL PlayMetaFileRecord( // 显示图元文件中的记录
    HDC hdc,                      // DC 句柄
    LPHANDLETABLE lpHandleTable,   // 元文件句柄表
    CONST ENHMETARECORD *lpMetaRecord, // 元文件记录
    UINT nHandles                  // 句柄计数
);
BOOL EnumMetaFile( // 枚举元文件
    HDC hdc,                      // DC 句柄
    HMETAFILE hmf,                // 元文件句柄
    MFENUMPROC lpMetaFunc,        // 回调函数
    LPARAM lParam                 // 可选数据
);
int CALLBACK EnumMetaFileProc( // 枚举元文件处理函数
    // 用户自己定义，由 EnumMetaFile 函数回调
    HDC hDC,                      // DC 句柄
    HANDLETABLE *lpHTable,         // 元文件句柄表
    METARECORD *lpMFR,            // 元文件记录
    int nObj,                     // 对象计数
    LPARAM lpClientData           // 可选数据
);

```

这些函数的使用方法，参见下面“(2) EMF”中的对应内容之例。

(2) EMF

主要 SDK 函数有：

```

HDC CreateEnhMetaFile( // 创建图元文件
    HDC hdcRef,                  // 参考 DC 的句柄
    LPCTSTR lpFilename,          // 文件名 (lpFilename=NULL 时，创建内存图元文件)
    CONST RECT* lpRect,           // 边界矩形 (单位为 0.01 毫米)
    LPCTSTR lpDescription        // 描述串
)

```

```

);
HENHMETAFILE CloseEnhMetaFile(HDC hdc); // 关闭图元文件
HENHMETAFILE GetEnhMetaFile(LPCTSTR lpszMetaFile); // 装入图元文件
BOOL PlayEnhMetaFile( // 显示图元文件
    HDC hdc,           // DC 句柄
    HENHMETAFILE hemf, // 增强型图元文件的句柄
    CONST RECT *lpRect // 边界矩形 (逻辑单位)
);
BOOL DeleteEnhMetaFile(HENHMETAFILE hemf); // 删除内存图元文件
                                         // 或删除磁盘图文件的句柄
DWORD GetLastError(void); // 获取最后一次图元文件操作的错误代码

```

例如：

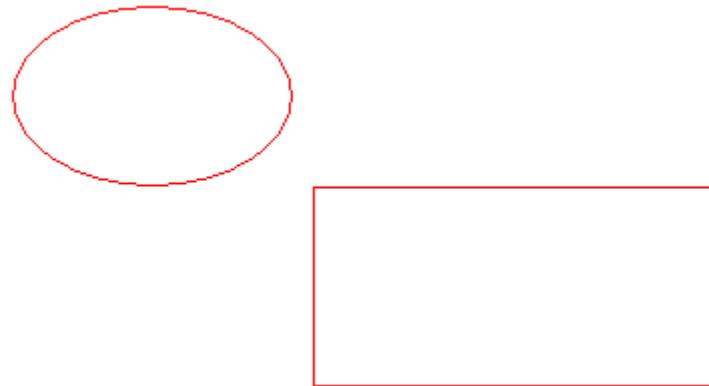
```

// 创建图元文件
// 获取 HDC 与屏幕大小
HDC hdcRef = GetDC()->m_hDC;
int iWidthMM = GetDeviceCaps(hdcRef, HORZSIZE); // 屏幕宽 (毫米)
int iHeightMM = GetDeviceCaps(hdcRef, VERTSIZE); // 屏幕高 (毫米)
int iWidthPels = GetDeviceCaps(hdcRef, HORZRES); // 屏幕宽 (像素)
int iHeightPels = GetDeviceCaps(hdcRef, VERTRES); // 屏幕高 (像素)
// 获取客户区大小，并将像素坐标转换为 0.01 毫米坐标
RECT rect;
::GetClientRect(this->m_hWnd, &rect); // 像素坐标
//rect.left = (rect.left * iWidthMM * 100)/iWidthPels;
//rect.top = (rect.top * iHeightMM * 100)/iHeightPels;
rect.right = (rect.right * iWidthMM * 100)/iWidthPels;
rect.bottom = (rect.bottom * iHeightMM * 100)/iHeightPels;
// 创建 EMF 文件
HDC hdcMeta = CreateEnhMetaFile(hdcRef, L"test.emf", &rect, NULL);
// 绘制图形
SelectObject(hdcMeta, CreatePen(PS_SOLID, 0, RGB(255, 0, 0)));
Ellipse(hdcMeta, 10, 10, 150, 100);
Rectangle(hdcMeta, 160, 100, 360, 200);
// 关闭图元文件
HENHMETAFILE hemf = CloseEnhMetaFile(hdcMeta);

// 装入并显示图元文件
HENHMETAFILE hemf = GetEnhMetaFile(L"test.emf");
RECT rect;
::GetClientRect(this->m_hWnd, &rect);
PlayEnhMetaFile(GetDC()->m_hDC, hemf, &rect);
DeleteEnhMetaFile(hemf);

```

输出结果为：



显示图元文件 test.emf

其他常用函数有：

```
UINT GetEnhMetaFileHeader( // 获取元文件头
    // 缓冲区参数为 NULL 时返回头大小，否则返回拷贝到缓冲区中的字节数
    HENHMETAFILE hemf,           // 元文件句柄
    UINT cbBuffer,                // 缓冲区大小
    LPENHMETAHEADER lpemh // 数据缓冲区
);
BOOL PlayEnhMetaFileRecord( // 显示元文件中的记录
    HDC hdc,                      // DC 句柄
    LPHANDLETABLE lpHandleTable,   // 元文件句柄表
    CONST ENHMETARECORD *lpEnhMetaRecord, // 元文件记录
    UINT nHandles                 // 句柄计数
);
BOOL EnumEnhMetaFile( // 枚举图元文件
    // 遍历图元文件，将每个记录交给回调函数处理
    HDC hdc,                      // DC 句柄
    HENHMETAFILE hemf,           // 元文件句柄
    ENHMFENUMPROC lpEnhMetaFunc, // 回调函数
    LPVOID lpData,                // 回调函数数据
    CONST RECT *lpRect            // 边界矩形（逻辑单位）
);
int CALLBACK EnhMetaFileProc( // 元文件处理函数
    // 用户自己定义，由 EnumEnhMetaFile 函数回调
    // 返回 FALSE(0)时终止遍历，返回 TRUE(非 0)时继续遍历
    HDC hDC,                     // DC 句柄
    HANDLETABLE *lpHTable,        // 元文件句柄表
    CONST ENHMETARECORD *lpEMFR,  // 元文件记录
    int nObj,                    // 对象计数
    LPARAM lpData                // 可选数据
);
```

例如：（显示指定记录）

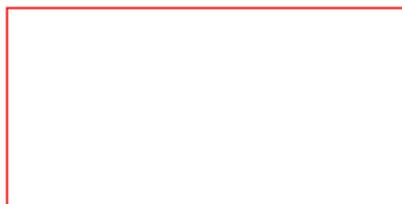
```

// 自定义回调函数（用于处理图元文件中的每个记录）
int CALLBACK EnhMetaFileProc(HDC hDC, HANDLETABLE *lpHTable,
    CONST ENHMETARECORD *lpEMFR, int nObj, LPARAM lpData) {
    /*if (lpEMFR->iType == EMR_CREATEPEN // 创建和选入红色笔
     || lpEMFR->iType == EMR_SELECTOBJECT)
     PlayEnhMetaFileRecord(hDC, lpHTable, lpEMFR, nObj);*/
    if (lpEMFR->iType == EMR_RECTANGLE) { // 画矩形
        PlayEnhMetaFileRecord(hDC, lpHTable, lpEMFR, nObj);
        return FALSE; // 终止遍历
    }
    return TRUE; // 继续遍历
}

// 装入图元文件，枚举（显示）图元文件（中的记录）
HENHMETAFILE hemf = GetEnhMetaFile(L"test.emf");
RECT rect;
::GetClientRect(this->m_hWnd, &rect);
EnumEnhMetaFile(GetDC()->m_hDC, hemf, // 遍历元文件中的每个记录
    ENHMFENUMPROC(&EnhMetaFileProc), NULL, &rect); // 调用回调函数处理
DeleteEnhMetaFile(hmf);

```

输出结果为：



不绘制椭圆



而且不创建和选入红色笔

显示图元文件 test.emf 中的记录

(3) 转换

可以利用 SDK 函数，对 WMF 和 EMF 图元文件进行相互转换。

● WMF → EMF

可以依次利用下面的 3 个 SDK 函数：

```

UINT GetMetaFileBitsEx( // 获取 WMF 位数据
    // 缓冲区参数为 NULL 时返缓冲区大小，否则返回拷贝到缓冲区中的字节数
    HMETAFILE hmf, // WMF 元文件句柄
    UINT nSize, // 元文件大小
    LPVOID lpvData // 元文件数据
);

```

和

```

HENHMETAFILE SetWinMetaFileBits( // 用 WMF 位数据创建内存 EMF
    UINT cbBuffer,           // 缓冲区大小
    CONST BYTE *lpbBuffer,   // 元文件数据缓冲区
    HDC hdcRef,             // 参考 DC 的句柄
    CONST METAFILEPICT *lpmfp // 元文件图片的大小
);

```

及

```

HENHMETAFILE CopyEnhMetaFile( // 将 (内存) EMF 复制到指定 EMF 文件
    HENHMETAFILE hemfSrc,   // 增强型有文件的句柄
    LPCTSTR lpszFile // 文件名 (lpszFile=NULL 时, 复制到内存图元文件)
);

```

将一个 WMF 元文件转换为 EMF 元文件。其中：lpmfp 为结构

```

typedef struct tagMETAFILEPICT {
    LONG mm;
    LONG xExt;
    LONG yExt;
    HMETAFILE hMF;
} METAFILEPICT;

```

的指针，一般取为 NULL 即可。

例如：

```

HMETAFILE hwmf = GetMetaFile(L"test.wmf");
if (hwmf != NULL) {
    UINT size = 0;
    size = GetMetaFileBitsEx(hwmf, size, NULL);
    BYTE *buff = new BYTE[size];
    GetMetaFileBitsEx(hwmf, size, buff);
    HENHMETAFILE hemf = SetWinMetaFileBits(size, buff, GetDC()->m_hDC, NULL);
    if (hemf != NULL) {
        CopyEnhMetaFile(hemf, L"test1.emf");
        DeleteMetaFile(hwmf);
    }
    else MessageBox(L"Create memory EMF file fail!");
    DeleteEnhMetaFile(hemf);
}
else MessageBox(L"Load WMF file fail!");

```

● EMF → WMF

也可以依次利用下面的 3 个 SDK 函数：

```

UINT GetWinMetaFileBits( // 获取 EMF 位数据
    // 缓冲区参数为 NULL 时返回缓冲区大小, 否则返回拷贝到缓冲区中的字节数
    HENHMETAFILE hemf, // 增强型元文件句柄
    UINT cbBuffer,       // 缓冲区大小
    LPBYTE lpbBuffer,   // 文件缓冲区

```

```

INT fnMapMode,           // 映射模式
HDC hdcRef              // 参考 DC 的句柄
);
和
HMETAFILE SetMetaFileBitsEx( // 由数据创建内存 WMF 文件
    UINT nSize,           // WMF 文件大小
    CONST BYTE *lpData    // 元文件数据
);
及
HMETAFILE CopyMetaFile( // 将 (内存) WMF 复制到指定 WMF 文件
    HMETAFILE hmfSrc,   // handle to Windows-format metafile
    LPCTSTR lpszFile    // file name (lpszFile=NULL 时, 复制到内存图元文件)
);
将一个 EMF 元文件转换为 WMF 元文件。

```

例如:

```

HENHMETAFILE hemf = GetEnhMetaFile(L"test.emf");
if (hemf != NULL) {
    HDC hdcRef = GetDC()->m_hDC;
    UINT size = GetWinMetaFileBits(hemf, 0, NULL, MM_TEXT, hdcRef);
    BYTE *buf = new BYTE[size];
    GetWinMetaFileBits(hemf, size, buf, MM_TEXT, hdcRef);
    HMETAFILE hwmf = SetMetaFileBitsEx(size, buf);
    if (hwmf != NULL) {
        CopyMetaFile(hwmf, L"test1.wmf");
        DeleteMetaFile(hwmf);
    }
    else MessageBox(L"Create memory WMF file fail!");
    DeleteEnhMetaFile(hemf);
}
else MessageBox(L"Load EMF file fail!");

```

(4) 动态重绘

在视图类中定义类变量:

```

RECT rect0, rect1;
HDC hdcMeta;

```

在视图类的初始化函数 OnInitialUpdate 中, 创建内存 EMF 图元文件:

```

// 获取参考 HDC 与屏幕大小
HDC hdcRef = GetDC()->m_hDC;
int iWidthMM = GetDeviceCaps(hdcRef, HORZSIZE); // 屏幕宽 (毫米)
int iHeightMM = GetDeviceCaps(hdcRef, VERTSIZE); // 屏幕高 (毫米)
int iWidthPels = GetDeviceCaps(hdcRef, HORZRES); // 屏幕宽 (像素)

```

```

int iHeightPels = GetDeviceCaps(hdcRef, VERTRES); // 屏幕高（像素）
// 计算屏幕大小的 HIMETRIC 单位（0.01 毫米）值和逻辑单位（像素）值
rect0.left = 0;
rect0.top = 0;
rect0.right = iWidthMM * 100;
rect0.bottom = iHeightMM * 100;
rect1.left = 0;
rect1.top = 0;
rect1.right = iWidthPels;
rect1.bottom = iHeightPels;
// 创建 EMF 文件
hdcMeta = CreateEnhMetaFile(hdcRef, NULL, &rect0, NULL);

```

在视图类的 OnLButtonUp 等函数中，利用图元文件 DC 句柄，和各种 SDK 中的 GDI 设置与绘图函数，向图元文件添加各种绘图记录。如：

```

SelectObject(hdcMeta, pLinePen);
MoveToEx(hdcMeta, point1.x, point1.y, NULL);
LineTo(hdcMeta, point2.x, point2.y);
.....

```

在视图类的 OnDraw 函数中，关闭元文件，同时获取元文件句柄，播放元文件。然后再创建新的元文件，并将元文件中现有的记录，通过新的元文件的 DC 播放，加入到新元文件中，最后删除老元文件的句柄。如：

```

HENHMETAFILE hemf = CloseEnhMetaFile(hdcMeta); // 关闭元文件
PlayEnhMetaFile(GetDC()->m_hDC, hemf, &rect1); // 播放元文件
// 创建新的内存 EMF 文件
hdcMeta = CreateEnhMetaFile(pDC->m_hDC, NULL, &rect0, NULL);
PlayEnhMetaFile(hdcMeta, hemf, &rect1); // 将老元文件的纪录加入到新元文件中
DeleteEnhMetaFile(hemf); // 删除元文件句柄

```

可以在视图类的某个消息响应函数中利用 SDK 函数：

```
HMETAFILE CopyMetaFile(HMETAFILE hmfSrc, LPCTSTR lpszFile);
```

或

```
HENHMETAFILE CopyEnhMetaFile(HENHMETAFILE hemfSrc, LPCTSTR lpszFile);
```

来将内存中的图元文件保存到磁盘文件中。如：

```

HENHMETAFILE hemf = CloseEnhMetaFile(hdcMeta); // 关闭元文件
CopyEnhMetaFile(hemf, L"draw.emf"); // 复制到磁盘文件

```

最后，在视图类的析构函数中，关闭并删除图元文件。例如：

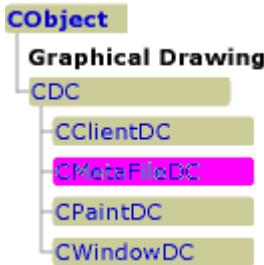
```

HENHMETAFILE hemf = CloseEnhMetaFile(hdcMeta);
DeleteEnhMetaFile(hemf);

```

3) MFC 的 CMetaFileDC 类

以上的讨论是基于 SDK 的，现在我们来介绍 MFC 对图元文件的封装类 CMetaFileDC，它是 CDC 的子类：



CMetaFileDC 类及其基类和兄弟类

CMetaFileDC 类很简单，只有一个缺省构造函数——CMetaFileDC、两个具体创建函数——Create（创建 WMF）和 CreateEnhanced（创建 EMF）、两个关闭函数——Close（关闭 WMF）和 CloseEnhanced（关闭 EMF）。它主要是使用其父类 CDC 的成员函数。

(1) 创建 CMetaFileDC 对象

创建 CMetaFileDC 对象分两步进行，首先调用 CMetaFileDC 类的唯一（缺省）构造函数：

CMetaFileDC();

来构造一个空对象，然后再调用该类的 Create 成员函数：

BOOL Create(LPCTSTR lpszFilename = NULL);

来创建一个 WMF 图元文件对象。或 CreateEnhanced 成员函数：

BOOL CreateEnhanced(CDC* pDCRef, LPCTSTR lpszFileName,
LPCRECT lpBounds, LPCTSTR lpszDescription);

来创建一个 EMF 图元文件对象。

其中：

- pDCRef —— 参考 DC，一般取为当前视图类的 m_hDC 数据成员。为 NULL 时，使用当前显示设备作为参考 DC；
- lpszFilename —— 图元文件名输入参数，如果为 NULL，则创建内存图元文件；
- lpBounds —— 边界矩形（单位是 0.01 毫米），可以是 RECT 结构或 CRect 对象的指针。为 NULL 时，取包含用户图形的最小矩形。
- lpszDescription —— 描述字符串，一般包含应用程序名和图名，也可以为 NULL。

例如：

```
CMetaFileDC metaDC; // 构造元文件 DC 空对象  
// metaDC.Create(NULL); // 创建内存 WMF 元文件 DC  
// metaDC.Create(L"test.wmf"); // 创建 WMF 元文件 DC  
// metaDC.CreateEnhanced(NULL, NULL, NULL, NULL); // 创建内存 EMF 元文件 DC  
// 获取屏幕大小  
HDC hdcRef = GetDC()->m_hDC;
```

```

int iWidthMM = GetDeviceCaps(hdcRef, HORZSIZE); // 屏幕宽 (毫米)
int iHeightMM = GetDeviceCaps(hdcRef, VERTSIZE); // 屏幕高 (毫米)
int iWidthPels = GetDeviceCaps(hdcRef, HORZRES); // 屏幕宽 (像素)
int iHeightPels = GetDeviceCaps(hdcRef, VERTRES); // 屏幕高 (像素)
// 获取客户区大小，并将像素坐标转换为 0.01 毫米坐标
RECT rect;
GetClientRect(&rect); // 像素坐标
rect.right = (rect.right * iWidthMM * 100)/iWidthPels;
rect.bottom = (rect.bottom * iHeightMM * 100)/iHeightPels;
// 创建 EMF 元文件 DC
metaDC.CreateEnhanced(GetDC(), L"test.emf", &rect, L"MFCDraw MixGraphics");

```

(2) 添加绘图记录

因为 CMetaFileDC 是 CDC 的派生类，可以像用普通 CDC 对象一样地，使用 CMetaFileDC 来设置绘图环境和执行绘图指令。例如：

```

metaDC.SelectObject(pLinePen);
metaDC.MoveTo(point1);
metaDC.LineTo(point2);

```

(3) 播放图元文件

为了播放图元文件，必须先关闭图元文件 DC，这可交由 CMetaFileDC 类的成员函数：

```

HMETAFILE Close(); // 关闭 WMF 图元文件 DC，出错返回 NULL
HENHMETAFILE CloseEnhanced(); // 关闭 EMF 图元文件 DC，出错返回 NULL

```

来完成。然后用它们的返回值（图元文件的句柄）来调用 CDC 类的成员函数：

```

BOOL PlayMetaFile(HMETAFILE hMF); // 播放 WMF 图元文件
// 播放 EMF 图元文件
BOOL PlayMetaFile(HENHMETAFILE hEnhMetaFile, LPCRECT lpBounds);

```

（该函数有一点不听话，可能需要使用下面的 SDK 函数：

```
BOOL PlayMetaFile(HDC hdc, HMETAFILE hmf); // 播放 WMF 图元文件
```

```
BOOL PlayEnhMetaFile() // 播放 EMF 图元文件
```

```

HDC hdc, // DC 句柄
HENHMETAFILE hemf, // 增强型图元文件的句柄
CONST RECT *lpRect // 边界矩形 (逻辑单位)
);

```

来代替）

来播放图元文件（重画绘图记录）。其中的 lpBounds 为边界矩形，逻辑单位（缺省为像素）。

在使用完图元文件句柄后，应该调用 SDK 函数：

```
BOOL DeleteMetaFile(HMETAFILE hmf);
```

来删除它（对磁盘元文件它只删除句柄，对内存元文件它则删除内存中元文件的所有内容）。

例如：

```

HENHMETAFILE hemf = metaDC.CloseEnhanced();
// pDC->PlayMetaFile(hemf, &rectPixel); // 此函数不听话

```

```

PlayEnhMetaFile(GetDC()->m_hDC, hemf, &rect1); // SDK 函数
DeleteEnhMetaFile(hemf);

```

说明：MFC 所封装的图元文件 DC 类 CMetaFileDC，只是用于创建新图元文件，并添加绘图记录。虽然，可以播放自己创建的图元文件，但是却不能播放已经存在的磁盘图元文件。解决办法是，直接使用 SDK 函数：

```

HENHMETAFILE GetEnhMetaFile(LPCTSTR lpszMetaFile); // 装入图元文件
BOOL PlayEnhMetaFile( // 显示图元文件
    HDC hdc, // DC 句柄
    HENHMETAFILE hemf, // 增强型图元文件的句柄
    CONST RECT *lpRect // 边界矩形（逻辑单位）
);
UINT GetEnhMetaFileHeader( // 获取元文件头
    // 缓冲区参数为 NULL 时返回缓冲区大小，否则返回拷贝到缓冲区中的字节数
    HENHMETAFILE hemf, // 增强型图元文件的句柄
    UINT cbBuffer, // 缓冲区大小
    LPENHMETAHEADER lpemh // 数据缓冲区
);

```

例如：

```

HENHMETAFILE hemf = GetEnhMetaFile(L"test.emf");
UINT size = GetEnhMetaFileHeader(hemf, 0, NULL);
ENHMETAHEADER *emHeader = (ENHMETAHEADER *)malloc(size);
GetEnhMetaFileHeader(hemf, size, emHeader);
RECTL rectl = emHeader->rclBounds; // 边界矩形
RECT rect = {rectl.left, rectl.top, rectl.right, rectl.bottom};
PlayEnhMetaFile(GetDC()->m_hDC, hemf, &rect);
DeleteEnhMetaFile(hemf);

```

(4) 动态重放元文件

在视图类中定义类变量：

```

RECT rectHimm, rectPixel;
CMetaFileDC metaDC;

```

在视图类的初始化函数 OnInitialUpdate 中，创建 EMF 元文件 DC：

```

// 获取参考 HDC 与屏幕大小
HDC hdcRef = GetDC()->m_hDC;
int iWidthMM = GetDeviceCaps(hdcRef, HORZSIZE); // 屏幕宽（毫米）
int iHeightMM = GetDeviceCaps(hdcRef, VERTSIZE); // 屏幕高（毫米）
int iWidthPels = GetDeviceCaps(hdcRef, HORZRES); // 屏幕宽（像素）
int iHeightPels = GetDeviceCaps(hdcRef, VERTRES); // 屏幕高（像素）
// 计算屏幕大小的 HIMETRIC 单位（0.01 毫米）值和逻辑单位（像素）值
rectHimm.left = 0;
rectHimm.top = 0;

```

```

rectHimm.right = iWidthMM * 100;
rectHimm.bottom = iHeightMM * 100;
rectPixel.left = 0;
rectPixel.top = 0;
rectPixel.right = iWidthPels;
rectPixel.bottom = iHeightPels;
// 创建内存 EMF 文件 DC
metaDC.CreateEnhanced(GetDC(), NULL, &rectHimm, NULL);

```

在视图类的 OnLButtonUp 等函数中, 利用图元文件 DC, 向图元文件添加各种绘图记录。如:

```

metaDC.SelectObject(pLinePen);
metaDC.MoveTo(point1);
metaDC.LineTo(point2);
.....

```

在视图类的 OnDraw 函数中, 关闭元文件 DC, 同时获取元文件句柄, 播放元文件。然后再创建新的元文件 DC, 并将老元文件中现有的记录, 通过新的元文件 DC 的播放, 加入到新元文件中, 最后删除老元文件的句柄。如:

```

HENHMETAFILE hemf = metaDC.CloseEnhanced(); // 关闭元文件 DC, 获取句柄
PlayEnhMetaFile(GetDC()->m_hDC, hemf, &rectPixel); // 播放元文件 (SDK 函数)
// pDC->PlayMetaFile(hemf, &rectPixel); // 此函数不听话
metaDC.CreateEnhanced(GetDC(), NULL, &rectHimm, NULL); // 创建新元文件 DC
metaDC.PlayMetaFile(hemf, &rectPixel); // 将老元文件的纪录加入到新元文件中
DeleteEnhMetaFile(hemf); // 删除元文件句柄 (SDK 函数)

```

可以在视图类的某个消息响应函数中利用 SDK 函数:

```
HMETAFILE CopyMetaFile(HMETAFILE hmfSrc, LPCTSTR lpszFile);
```

或

```
HENHMETAFILE CopyEnhMetaFile(HENHMETAFILE hemfSrc, LPCTSTR lpszFile);
```

来将内存中的图元文件保存到磁盘文件中。例如:

```

HENHMETAFILE hemf = CloseEnhMetaFile(hdcMeta); // 关闭元文件, 获取句柄
CopyEnhMetaFile(hemf, L"draw.emf"); // 复制到磁盘文件 (SDK 函数)
DeleteEnhMetaFile(hemf); // 删除元文件句柄 (SDK 函数)

```

最后, 在视图类的析构函数中, 关闭图元文件 DC 并删除图元文件句柄。例如:

```

HENHMETAFILE hemf = metaDC.CloseEnhanced();
DeleteEnhMetaFile(hemf);

```

(5) 播放图元文件的记录

CMetaFileDC 类和其父类 CDC 中都没有播放图元文件记录的成员函数, 但是可以利用 CMetaFileDC 类的 Close 或 CloseEnhanced 成员函数所获得的图元文件句柄, 再调用 SDK 的有关函数: (类似于 2) (2))

```

BOOL PlayEnhMetaFileRecord( // 显示元文件中的记录
    HDC hdc,                                // DC 句柄
    LPHANDLETABLE lpHandleTable,             // 元文件句柄表
    CONST ENHMETARECORD *lpEnhMetaRecord,   // 元文件记录
    UINT nHandles                           // 句柄计数
);
BOOL EnumEnhMetaFile( // 枚举图元文件
    // 遍历图元文件，将每个记录交给回调函数处理
    HDC hdc,                                // DC 句柄
    HENHMETAFILE hemf,                     // 元文件句柄
    ENHMFENUMPROC lpEnhMetaFunc,           // 回调函数
    LPVOID lpData,                          // 回调函数数据
    CONST RECT *lpRect                      // 边界矩形（逻辑单位）
);
int CALLBACK EnhMetaFileProc( // 元文件处理函数
    // 用户自己定义，由 EnumEnhMetaFile 函数回调
    // 返回 FALSE(0)时终止遍历，返回 TRUE(非 0)时继续遍历
    HDC hDC,                                // DC 句柄
    HANDLETABLE *lpHTable,                  // 元文件句柄表
    CONST ENHMETARECORD *lpEMFR,            // 元文件记录
    int nObj,                               // 对象计数
    LPARAM lpData                           // 可选数据
);

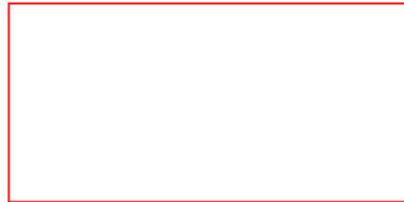
```

例如：（显示指定记录）

```

// 创建 EMF 文件 DC
CMetaFileDC metaDC;
metaDC.CreateEnhanced(GetDC(), L"test.emf", &rectHimm, NULL);
// 绘制图形
metaDC.SelectObject(new CPen(PS_SOLID, 0, RGB(255, 0, 0)));
metaDC.Ellipse(10, 10, 150, 100);
metaDC.Rectangle(160, 100, 360, 200);
// 关闭图元文件
HENHMETAFILE hemf = metaDC.CloseEnhanced();
// 播放图元文件
PlayEnhMetaFile(GetDC()->m_hDC, hemf, &rectPixel);
// 删除图元文件
DeleteEnhMetaFile(hemf);

```



播放图元文件 test.emf

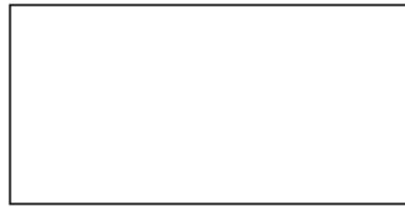
```
// 自定义回调函数（用于处理图元文件中的每个记录）
int CALLBACK EnhMetaFileProc(HDC hDC, HANDLETABLE *lpHTable,
    CONST ENHMETARECORD *lpEMFR, int nObj, LPARAM lpData) {
/*if (lpEMFR->iType == EMR_CREATEPEN // 创建和选入红色笔
    || lpEMFR->iType == EMR_SELECTOBJECT)
    PlayEnhMetaFileRecord(hDC, lpHTable, lpEMFR, nObj);*/
if (lpEMFR->iType == EMR_RECTANGLE) { // 画矩形
    PlayEnhMetaFileRecord(hDC, lpHTable, lpEMFR, nObj);
    return FALSE; // 终止遍历
}
return TRUE; // 继续遍历
}

// 装入图元文件，枚举（显示）图元文件（中的记录）
EnumEnhMetaFile(GetDC()->m_hDC, hemf, // 遍历元文件中的每个记录
    ENHMFENUMPROC(&EnhMetaFileProc), NULL, &rectPixel); // 调用回调函数处理
DeleteEnhMetaFile(hemf); // 删除图元文件
```

输出结果为：



不绘制椭圆



而且不创建和选入红色笔

显示图元文件 test.emf 中的记录

4) GDI+中的图元文件类

在 GDI+中，图元文件对应的类为 `Metafile`，它是 `Image` 类的派生类。GDI+的 `Metafile` 类支持三种类型的图元文件：仅 EMF 类型、仅 EMF+类型、EMF 及 EMF+双重类型（缺省值）。它们对应于枚举类型：

```

typedef enum {
    EmfTypeEmfOnly = MetafileTypeEmf, // 仅 EMF 类型
    EmfTypeEmfPlusOnly = MetafileTypeEmfPlusOnly, // 仅 EMF+类型
    EmfTypeEmfPlusDual = MetafileTypeEmfPlusDual // EMF 及 EMF+双重类型
} EmfType;

```

(1) Metafile 类的构造函数

Metafile 类有 13 个构造函数:

```

// 文件型
Metafile(const WCHAR *filename);
Metafile(const WCHAR *fileName, HDC referenceHdc, EmfType type =
        EmfTypeEmfPlusDual, const WCHAR *description = NULL);
Metafile(const WCHAR *fileName, HDC referenceHdc, const Rect &frameRect,
        MetaFileFrameUnit frameUnit = MetafileFrameUnitGdi, EmfType type =
        EmfTypeEmfPlusDual, const WCHAR *description = NULL);
Metafile(const WCHAR *fileName, HDC referenceHdc, const RectF &frameRect,
        MetafileFrameUnit frameUnit = MetafileFrameUnitGdi, EmfType type =
        EmfTypeEmfPlusDual, const WCHAR *description = NULL);

// 流型
Metafile(IStream *stream);
Metafile(IStream *stream, HDC referenceHdc, EmfType type = EmfTypeEmfPlusDual,
        const WCHAR *description = NULL);
Metafile(IStream *stream, HDC referenceHdc, const Rect &frameRect,
        MetafileFrameUnit frameUnit = MetafileFrameUnitGdi, EmfType type =
        EmfTypeEmfPlusDual, const WCHAR *description = NULL);
Metafile(IStream *stream, HDC referenceHdc, const RectF &frameRect,
        MetafileFrameUnit frameUnit = MetafileFrameUnitGdi, EmfType type =
        EmfTypeEmfPlusDual, const WCHAR *description = NULL);

// DC 句柄型
Metafile(HDC referenceHdc, EmfType type = EmfTypeEmfPlusDual, const WCHAR
        *description = NULL);
Metafile(HDC referenceHdc, const Rect &frameRect, MetafileFrameUnit frameUnit =
        MetafileFrameUnitGdi, EmfType type = EmfTypeEmfPlusDual, const WCHAR
        *description = NULL);
Metafile(HDC referenceHdc, const RectF &frameRect, MetaFileFrameUnit frameUnit =
        MetafileFrameUnitGdi, EmfType type = EmfTypeEmfPlusDual, const WCHAR
        *description = NULL);

// WMF/EMF 句柄型
Metafile(HENHMETAFILE hEmf, BOOL deleteEmf = FALSE);
Metafile(HMETAFILE hWmf, const WmfPlaceableFileHeader *wmfPlaceableFileHeader,
        BOOL deleteWmf = FALSE);

```

其中用到的枚举类型有:

```
typedef enum {
```

```

MetafileFrameUnitPixel = UnitPixel, // 象素
MetafileFrameUnitPoint = UnitPoint, // 点
MetafileFrameUnitInch = UnitInch, // 英寸
MetafileFrameUnitDocument = UnitDocument, // 文档
MetafileFrameUnitMillimeter = UnitDocument + 1, // 毫米
MetafileFrameUnitGdi = UnitDocument + 2 // GDI+单位数目

} MetafileFrameUnit;

typedef struct {
    UINT32 Key; // 键
    INT16 Hmf; //
    PWMFRect16 BoundingBox; // 边界盒
    INT16 Inch; // 英寸
    UINT32 Reserved; // 保留
    INT16 Checksum; // 检测和
} WmfPlaceableFileHeader;

```

其中，最简单常用的构造函数是：// 不带 DC 参数，只能用于打开已经存在的元文件

```
Metafile(const WCHAR *filename);
```

它由文件名来构造元文件对象。例如：

```
Metafile mf(L"yyy.emf");
```

常用且完整的构造函数是：// 带 DC 参数，只用于创建新图元文件

```
Metafile(const WCHAR *fileName, HDC referenceHdc, EmfType type =
EmfTypeEmfPlusDual, const WCHAR *description = NULL);
```

它可以指定元文件类型，并加上描述串。例如：

```
Metafile mf(L"yyy.emf", GetDC()->m_hDC, MetafileTypeEmf, L"阴阳鱼");
```

另一个较为常用的构造函数是：

```
Metafile(HDC referenceHdc, EmfType type = EmfTypeEmfPlusDual, const WCHAR
*description = NULL);
```

它用于构造内存元文件。这些内存元文件构造函数还有对应的流构造函数版本。

(2) Metafile 类的成员函数

Metafile 类的其他成员函数有：

```

Status PlayRecord(EmfPlusRecordType recordType, UINT flags, UINT dataSize, const
BYTE *data); // 显示元文件记录，需要与 Graphics 类的 EnumerateMetafile
// 函数及用户自定义的回调函数配套使用（似 GDI 的）
static UINT EmfToWmfBits(HENHMETAFILE hemf, UINT cbData16, LPBYTE pData16,
INT iMapMode, EmfToWmfBitsFlags eFlags); // 用于 EMF 到 WMF 的转换
HENHMETAFILE GetHENHMETAFILE(VOID); // 可用于 EMF 的 SDK 函数
// 获取和设置底层光栅限制，用于减少刷空间大小
UINT GetDownLevelRasterizationLimit(VOID);
Status SetDownLevelRasterizationLimit(UINT metafileRasterizationLimitDpi);
// 获取元文件头
Status GetMetafileHeader(MetafileHeader *header) const;

```

```

static Status GetMetafileHeader(const WCHAR *filename, MetafileHeader *header);
static Status GetMetafileHeader(IStream *stream, MetafileHeader *header);
static Status GetMetafileHeader(HENHMETAFILE *hEmf, MetafileHeader *header);
static Status GetMetafileHeader(HMETAFILE hWmf, const WmfPlaceableFileHeader
    *wmfPlaceableFileHeader, MetafileHeader *header);

```

(3) MetafileHeader 类

MetafileHeader 类用于获取元文件的各种信息。该类无构造函数（可以通过 Metafile 类的成员函数 GetMetafileHeader 来获取该类的对象），但有许多其他成员函数：

```

BOOL IsWmf(VOID); // 判断是否是 WMF 元文件
BOOL IsEmf(VOID) const; // 判断是否是 EMF 元文件
BOOL IsEmfPlus(VOID) const; // 判断是否是 EMF+元文件
BOOL IsEmfOrEmfPlus(VOID) const; // 判断是否是 EMF 或 EMF+元文件
BOOL IsEmfPlusOnly(VOID) const; // 判断是否只是 EMF+元文件
BOOL IsEmfPlusDual(VOID) const; // 判断是否是 EMF 和 EMF+双重元文件
MetafileType GetType(VOID); // 获取元文件类型
UINT GetVersion(VOID); // 获取版本
UINT GetMetafileSize(VOID); // 获取元文件大小
UINT GetEmfPlusFlags(VOID); // 获取 EMF+标志位
const METAHEADER *GetWmfHeader(VOID) const; // 获取 WMF 头
const ENHMETAHEADER3 *GetEmfHeader(VOID) const; // 获取 EMF 头
void GetBounds(Rect *rect); // 获取边界矩形
REAL GetDpiX(VOID); // 获取水平分辨率 DPI
REAL GetDpiY(VOID); // 获取垂直分辨率 DPI
BOOL IsDisplay(VOID) const; // 判断是否由采用视频 DC 记录
BOOL IsWmfPlaceable(VOID) const; // 判断是否是可重定位的 WMF

```

和数据成员：

```

REAL DpiX; // 存储在关联元文件中图象的用每英寸点表示的水平分辨率
REAL DpiY; // 存储在关联元文件中图象的用每英寸点表示的垂直分辨率
UINT EmfPlusFlags; // 指定与元文件关联的 EMF+标志值
INT EmfPlusHeaderSize; // EMF+头的字节大小
INT X; // 元文件的最左边的坐标值
INT Y; // 元文件的最上边的坐标值
INT Width; // 存储在关联元文件中图象的用象素表示的宽度
INT Height; // 存储在关联元文件中图象的用象素表示的高度
INT LogicalDpiX; // 用每英寸点表示的逻辑水平分辨率
INT LogicalDpiY; // 用每英寸点表示的逻辑垂直分辨率
UINT Size; // 元文件的字节大小
MetafileType Type; // 指定元文件类型的 MetafileType 枚举元素
UINT Version; // 元文件的版本
union {WmfHeader, EmfHeader}; // METAHEADER 和 ENHMETAHEADER3 类型的联合

```

其中用到的元文件类型的枚举类型为：

```
enum MetafileType {
```

```

MetafileTypeInvalid,           // Invalid metafile
MetafileTypeWmf,              // Standard WMF
MetafileTypeWmfPlaceable,     // Placeable WMF
MetafileTypeEmf,               // EMF (not EMF+)
MetafileTypeEmfPlusOnly,       // EMF+ without dual, down-level records
MetafileTypeEmfPlusDual        // EMF+ with dual, down-level records
};

例如:

```

```

Metafile mf (L"test.emf");
MetafileHeader header;
mf.GetMetafileHeader(&header);
Rect rect;
header.GetBounds(&rect);
UINT size = header.GetMetafileSize();

```

5) 在 GDI+ 中使用图元文件

(1) 保存绘图

为了将绘图记录保存到图元文件中，需要先创建元文件对象，然后用该图元文件对象再来创建图形对象，最后调用图形类的各种绘图函数来向图元文件中添加绘图记录。

具体方法如下：

可以先使用 `Metafile` 类的用于创建新图元文件的构造函数（带 `DC` 参数的），如

```

Metafile(const WCHAR *fileName, HDC referenceHdc, EmfType type =
EmfTypeEmfPlusDual, const WCHAR *description = NULL);

```

来创建元文件对象。

然后使用 `Graphics` 类的构造函数（注意，`Metafile` 是 `Image` 的派生类）

```
Graphics(Image* image);
```

来创建图形对象。

最后调用各种图形类的图形设置、操作和绘制函数成员函数来向图元文件添加绘图记录。

例如：

```

Metafile *myMetafile = new Metafile(L"MyDiskFile.emf", GetDC()->m_hDC);
Graphics *myGraphics = new Graphics(myMetafile);
myGraphics->SetSmoothingMode(SmoothingModeAntiAlias);
myGraphics->RotateTransform(30);
// Create an elliptical clipping region.
GraphicsPath myPath;
myPath.AddEllipse(0, 0, 200, 100);
Region myRegion(&myPath);
myGraphics->SetClip(&myRegion);
Pen myPen(Color(255, 0, 0, 255));
myGraphics->DrawPath(&myPen, &myPath);
for(INT j = 0; j <= 300; j += 10) myGraphics->DrawLine(&myPen, 0, 0, 300 - j, j);

```

```
delete myGraphics;  
delete myMetafile;
```

(2) 重放绘图

可以先使用 `Metafile` 类的用于打开已有图元文件的构造函数（不带 DC 参数的），如
`Metafile(const WCHAR *filename);`

来创建元文件对象。

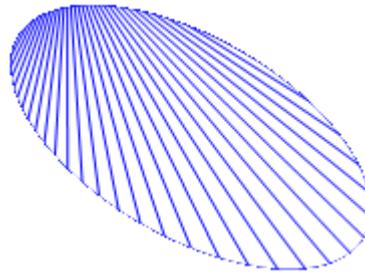
然后再调用 `Graphics` 类的各种 `DrawImage` 成员函数，如：

```
Status DrawImage(Image *image, INT x, INT y);
```

来重画图元文件中的所有绘图记录。例如：

```
Graphics *myGraphics = new Graphics(GetDC()->m_hDC);  
Metafile *myMetafile = new Metafile(L"MyDiskFile.emf");  
myGraphics->DrawImage(myMetafile, 0, 0);
```

输出结果为：



重放图元文件

另外，为了获取当前图元文件的边界矩形，可以先调用 `Metafile` 类的成员函数：

```
Status GetMetafileHeader(MetafileHeader *header) const;
```

来获取 `MetafileHeader` 对象，然后再用 `MetafileHeader` 类的成员函数：

```
void GetBounds(Rect *rect);
```

得到边界矩形。可用于 `Graphics` 类的 `DrawImage` 成员函数：

```
DrawImage(Image *image, const Rect &rect);
```

注意，如果用带 DC 参数的构造函数来创建 `Metafile` 对象，则会清空原图元文件（以便重新开始添加记录），不能用于图元文件的播放。

(3) 重画记录

可以利用 `Metafile` 类的成员函数

```
Status PlayRecord(EmfPlusRecordType recordType, UINT flags,  
UINT dataSize, const BYTE *data);
```

来重画图元文件中指定记录。与 EMF 中讨论的类似，该函数需要与 `Graphics` 类的枚举元文件成员函数（共有 12 个同名的重载函数），如：

```
Status EnumerateMetafile(const Metafile *metafile, const PointF &destPoint,  
EnumerateMetafileProc callback, VOID *callbackData = NULL,
```

```
    ImageAttributes *imageAttributes = NULL);
```

配套使用，该函数遍历图元文件的每个记录，并调用用户自定义的回调函数（该函数可以自己命名）

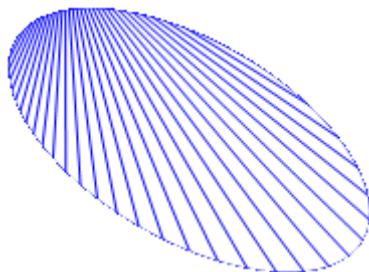
```
BOOL CALLBACK metaCallback(EmfPlusRecordType recordType, unsigned int flags,  
                           unsigned int dataSize, const unsigned char* pStr, void* callbackData);
```

对记录进行各种处理，包括使用元文件的成员函数 PlayRecord 来绘制（播放）记录。

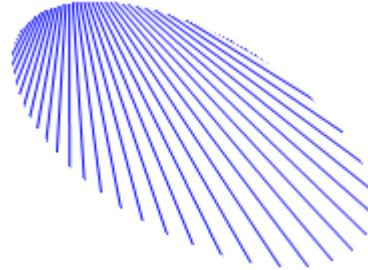
例如：

```
// 自定义的回调函数，不画椭圆路径（或/和取消椭圆区域剪裁）
```

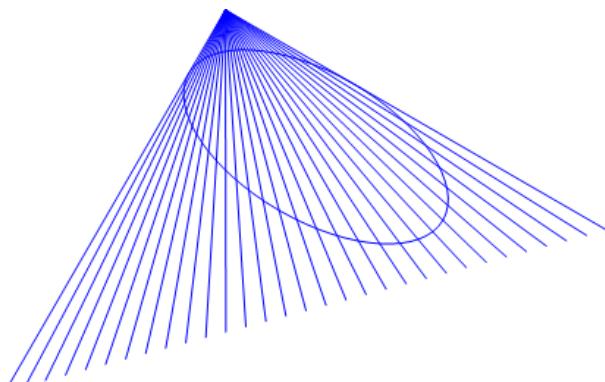
```
BOOL CALLBACK metaCallback(EmfPlusRecordType recordType, unsigned int flags,  
                           unsigned int dataSize, const unsigned char* pStr, void* callbackData) {  
    if (recordType == EmfPlusRecordTypeDrawPath  
        /*|| recordType == EmfPlusRecordTypeSetClipRegion*/)  
        return TRUE;  
    static_cast<Metafile*>(callbackData)->PlayRecord(recordType, flags, dataSize, pStr);  
    return TRUE;  
} // 注意：该函数不能作为视图类的成员函数，不然调用形式非常复杂  
// 枚举遍历图元文件  
Graphics graph(pDC->m_hDC);  
Metafile *pmf = new Metafile(L"MyDiskFile.emf");  
graph.EnumerateMetafile(mf, Point(0, 0), metaCallback, pmf);
```



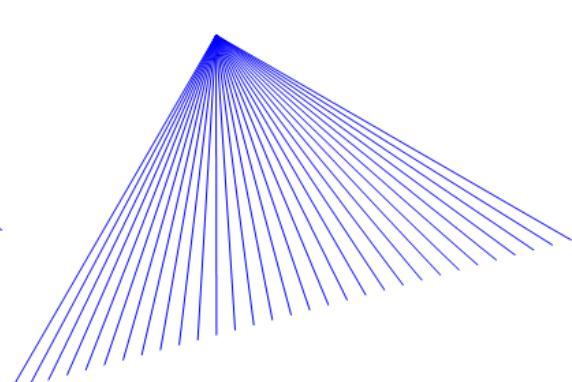
原始图元文件



不画椭圆路径



取消椭圆区域剪裁



不画椭圆路径并且取消椭圆区域剪裁

(4) 动态播放图元文件

因为 Metafile 类的构造函数被人为地分成了互不兼容、不能相互转换的两类：

- 带 DC 输入参数的构造函数——只能创建新图元文件，用于添加绘图记录，不能用于播放；
- 不带 DC 输入参数的构造函数——只能打开已有图元文件用于播放，不能用于添加绘图记录。

所以，在 GDI+ 中，想实现交互绘图时的窗口动态重画，非常困难。

虽然 Metafile 类有一个成员函数

HENHMETAFILE GetHENHMETAFILE(VOID);

可以用于获取图元文件的句柄，但经过我的实验发现，它只对使用不带 DC 输入参数的构造函数所创建的不能用于添加绘图记录的 Metafile 对象有效。

另外，虽说可以创建内存 Metafile 对象，但是 GDI+ 却没有提供任何办法（没有复制、保存、克隆等函数，父类 Image 的对应函数对写入型 Metafile 对象都是无效的），可将其保存到图元文件中。因为无法获得用于添加记录的图元文件的句柄，所以各种 SDK 函数也派不上用场。

因为除了帮助文档，几乎无资料可看，唯一的途径就是编码做试验。下面是我经过很长时间，好不容易才摸索出来的，一种可行的解决办法（但是很臭。你们可以寻找其他办法，如果有了更好的方法，请与大家共享）：

（说明：为了防止重画图元文件时，图形的位置有偏移或其大小发生变化，可以采用如下的构造函数：

```
Metafile(const WCHAR *fileName, HDC referenceHdc, const Rect &frameRect,
         MetaFileFrameUnit frameUnit = MetafileFrameUnitGdi, EmfType type =
         EmfTypeEmfPlusDual, const WCHAR *description = NULL);
```

来创建 Metafile 对象。其中的边框矩形，可以设置为屏幕大小，并使用像素单位。该边框同时还用于进行图元文件重画的 Graphics 类的 DrawImage 函数。）

在视图类中定义如下几个类变量：Metafile 对象及其对应的 Graphics 对象的指针、边框矩形、两个图元文件名的宽字符串数组（以便绕开 GDI+ 的文件锁定功能）、以及在这两个文件名中切换的整数。如：

```
Metafile *mf;
Graphics *mfGraph;
Rect rect0;
wchar_t *fns[2];
int fni;
```

在视图类的构造函数中，初始化部分类变量：

```
mf = NULL;
mfGraph = NULL;
fns[0] = L"draw.emf";
fns[1] = L"draw0.emf";
fni = 0;
```

在视图类的初始化函数 OnInitialUpdate 中，计算边框矩形、创建 Metafile 对象：

```
HDC hdcRef = GetDC()->m_hDC;
rect0.X = 0;
rect0.Y = 0;
rect0.Width = GetDeviceCaps(hdcRef, HORZRES);
rect0.Height = GetDeviceCaps(hdcRef, VERTRES);
mf = new Metafile(fns[fni], hdcRef, rect0, MetafileFrameUnitPixel);
mfGraph = new Graphics(mf);
```

在视图类的 OnLButtonUp 等函数中，利用图元文件所对应的图形对象，向图元文件添加各种绘图记录。如：

```
mfGraph->DrawLine(&Pen(Color::Green), p0.x, p0.y, point.x, point.y);
....
```

在视图类的 OnDraw 函数中，删除当前元文件对象(系统才会将元文件的内容写入磁盘)和对应的图形对象，打开该磁盘元文件并播放。然后，切换文件名，创建新的元文件对象和对应的图形对象，并将老元文件中现有的记录，通过新元文件所对应的图形对象的图像绘制，加入到新元文件中，最后删除老元文件的句柄。如：

```
delete mfGraph;
delete mf;
Metafile *mf0 = new Metafile(fns[fni]);
Graphics graph(pDC->m_hDC);
graph.DrawImage(mf0, rect0);
fni = !fni; // 相当于 if(fni) fni = 0; else fni = 1;
mf = new Metafile(fns[fni], pDC->m_hDC, rect0, MetafileFrameUnitPixel);
mfGraph = new Graphics(mf);
mfGraph->DrawImage(mf0, rect0);
delete mf0;
```

最后，在视图类的析构函数中，删除当前元文件对象(系统会将元文件的内容写入磁盘)和对应的图形对象。例如：

```
delete mfGraph;
delete mf;
```

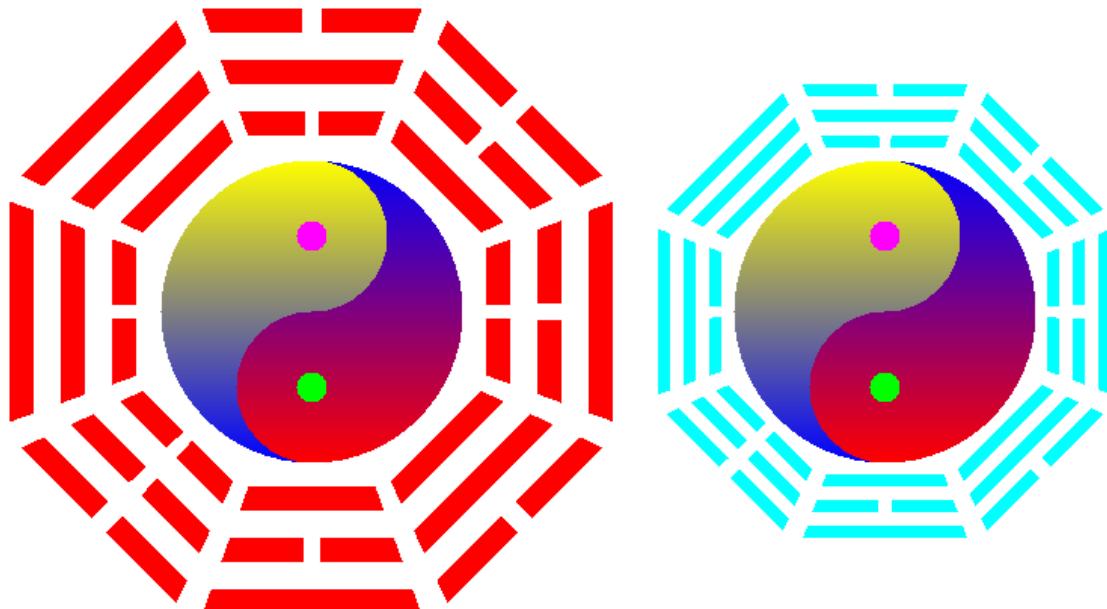
6.3 GDI+的托管代码编程

参考图书

- 周鸣杨、赵景亮. 精通 GDI+编程. 清华大学出版社, 2004 年 2 月. 16 开/463 页/42 元 (C++ / MFC)
- Mahesh Chand (韩江等译). GDI+图形程序设计. 电子工业出版社, 2005 年 3 月. 16 开/533 页/69 元 (C# / .NET)

作业

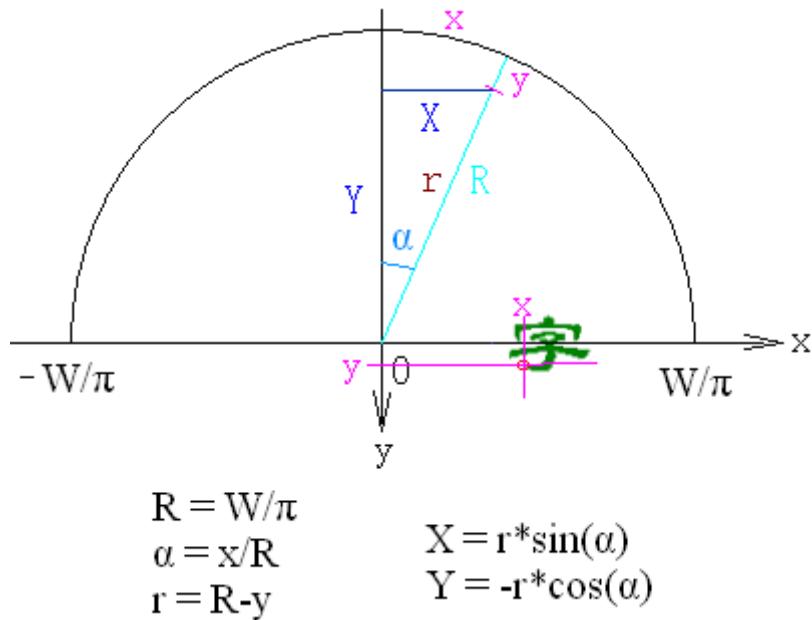
2. 实现课件中的各例。
3. 编写一个画填充正叶曲线的通用程序。可以指定叶片数、颜色、圆心和叶长(半径)等。
4. 编写一个绘制阴阳八卦图的通用程序。可以指定各种颜色、圆心和半径等等。形如：



5. 编写一个使用 GDI+接口的交互式绘图程序，实现 GDI+的所有基本功能和各种新增加的功能。(可以 2~3 周后交)
6. (选做) 编写若干(带不同类型输入参数的)画(线框和填充)弓弦和圆角矩形的 GDI+ 函数 (DrawChord、FillChord、DrawRoundRectangle 和 FillRoundRectangle)。
7. (选做) 绘制颜色枚举常量色块图。
8. (选做) 利用旋转变换编写一个时钟程序。(界面和功能自己设计)
9. 编写一个在圆周上绘制文本串的程序，输出效果类似于：(参见 6.2 的 6.3) 中的例子)



提示：(下面为圆周文本串的算法示意图，供大家参考)



半圆周文字算法示意图
(对全圆周, $R = W/2\pi$, 其中 W 为字符串路径的边界矩形宽度)

10. 编写一个图像程序, 实现上面介绍的各种图像操作与功能。
11. 将公爵动画用的 10 个 BMP 文件, 保存到一个多帧 TIFF 文件中。并实现 GIF 和 TIFF 动画。
12. 利用图元文件实现 MFC 和/或 GDI+交互绘图的重画功能。